

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！



OpenStack

设计与实现 (第2版)

英特尔开源技术中心 编著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

王庆：博士，OpenStack基金会个人独立董事，英特尔开源技术中心网络和存储部门开发经理，除OpenStack外，还致力于存储Ceph，网络OpenDaylight以及OPNFV的研发，《系统虚拟化：原理与实现》作者之一。

丁建峰：英特尔开源技术中心数据中心中国区软件研发经理，负责OpenStack等云计算项目的研发。

任桥伟：致力于Linux内核以及驱动的开发，著有《Linux内核修炼之道》《Linux那些事儿》系列。

陆连浩：Celiometer项目的Core Developer，此前长期从事Linux驱动、嵌入式系统开发工作。

翟纲：多年从事Linux内核、虚拟化的研究工作，并参与开源社区的开发。

贺永立：活跃于Nova、Neutron等项目，多年数据通讯、虚拟化以及Linux相关项目的开发经验。

徐贺杰：Nova项目的Core Developer。

程盈心：Nova社区的活跃贡献者，专注于分布式系统的分析与优化。

臧锐：目前致力于研究云计算网络子系统，多年操作系统内核、存储系统以及虚拟化系统的开发经验。

李晓燕：活跃于Cinder和Ceph项目，多年存储领域经验。

郭瑞景：从事网络与存储开发工作，活跃于Openstack，OpenDaylight，OPNFV等开源项目。

陈巍：KeyStone项目的Core Developer，OpenAttestation (OAT) 项目的主要贡献者之一。

杨林：从事云计算平台的开发工作，目前致力于从IaaS到PaaS，SaaS的演化以及混合云的研究。

乔立勇：Magnum和Zun项目的Core Developer，从事虚拟化和容器方向研发。

金运通：致力于Linux、虚拟化及云计算技术的开发。

杜永丰：活跃于OpenStack部署相关项目，多年从事操作系统内核、存储系统开发工作。

张磊：Searchlight项目的Core Developer。

冯少合：长期从事Linux、虚拟化及云计算技术相关开源项目的开发。

OpenStack

设计与实现 (第2版)

英特尔开源技术中心 编著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书是一本介绍 OpenStack 设计与实现原理的书。本书内容以 Newton 版本为基础, 覆盖了 OpenStack 的学习方法到设计与实现等各个方面内容, 致力于帮助读者形成 OpenStack 及其各个主要组件与项目的拓扑。

本书语言通俗易懂, 能够带领读者更为快速走入 OpenStack 的世界并作出自己的贡献。

本书适合希望能够参与 OpenStack 开发的读者, 也适合对 OpenStack 茫然的初学者, 以及有一定使用部署经验但是希望了解 OpenStack 实现原理的广大用户。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有, 侵权必究。

图书在版编目(CIP)数据

OpenStack 设计与实现 / 英特尔开源技术中心编著. —2 版. —北京: 电子工业出版社, 2017.5
ISBN 978-7-121-31199-4

I. ①O… II. ①英… III. ①计算机网络 IV. ①TP393

中国版本图书馆 CIP 数据核字(2017)第 065892 号

责任编辑: 徐津平

印 刷: 北京京科印刷有限公司

装 订: 三河市华成印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 33 字数: 712 千字

版 次: 2017 年 5 月第 1 版

印 次: 2017 年 5 月第 1 次印刷

印 数: 3000 册 定价: 99.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: 010-51260888-819, faq@phei.com.cn。

推荐序 1

It's impressive to see just how far OpenStack has come since its founding. With customers increasingly recognizing OpenStack maturity and a thriving ecosystem of developers and partners supporting the solution, it's clear that OpenStack has cemented its role as a leader in open cloud Infrastructure-as-a-Service.

Intel and China have played important roles in that growth. Intel as a strong contributor across OpenStack and related open source technologies; China as a critical source of customers, technical leadership and development talent. Recognizing the strength of this partnership, Intel has invested in the China technology ecosystem through our team working in the Intel Open Source Technology Center in China. This talented team authored the first edition of this book and is now proud to offer a second edition, reflecting the latest trends and updates from the OpenStack Newton release, including new projects (Murano, Searchlight, Kolla), deployment tools, and monitoring improvements as well as technologies and solutions (Ceph, containers) which integrate well with OpenStack.

We look forward to working with the China OpenStack community on the continued success of OpenStack.

Hillarie Prestopine

Intel Vice President and Director, Intel Open Source Technology Center

推荐序 2

Since its introduction in 2010, OpenStack has gained tremendous momentum with thousands of developers and a couple of thousand deployments. Today, OpenStack is considered the de-facto open source cloud Infrastructure-as-a-Service provisioning software for companies including Baidu, the European Organization for Nuclear Research (CERN), HP, Huawei, IBM, Intel, and Walmart, as well as many more Fortune 500 companies.

Intel strongly believes investments and engagements in open source help foster a strong ecosystem. The Open Source Technology Center at Intel is proud to be a leader in open source software across a wide range of technologies and market segments, including enterprise Linux and related technologies such as virtualization, data center and cloud software; embedded market segments and client Linux programs. Intel is one of the top 10 OpenStack contributors, with a focus on Nova, Horizon, Ironic, Ceilometer, Swift, and Cinder.

Increasingly established companies and new ventures are considering OpenStack to achieve savings through pooled compute, storage, and network resource utilization. China represents a vast talent resource pool, from young engineers groomed in universities to professionals in growing industries. The need for an OpenStack developer resource in Mandarin can help engineers learn about the cloud and OpenStack while speeding their ability to contribute to and influence OpenStack.

It is with immense joy that we present this book to the Chinese community. I hope you find it compelling and its content helps increase China's influence in OpenStack development, easing adoption and offering cost savings for new economic endeavors.

Imad Sousou

Intel Vice President and General Manager, Open Source Technology Center
Platinum Board Member, OpenStack Board of Directors

前言

至此落笔之际，OpenStack 问世几近 7 年，7 年的时间，对很多项目来说已经足够走过一个创建发展到没落的轮回，而对于 OpenStack，7 年的时间仍然远远不够让我们看到它最终能够达到的高度。

从哲学的辩证角度：今天的必然正是由之前一系列的偶然所决定的。2010 年的一个偶然，OpenStack 由 RackSpace 和美国国家航空航天局合作发布，于是随后的时间里无数公司与个人偶然初识 OpenStack 并深陷其中，而正是这些偶然相联合，从而决定了会有今天这样一本书，会有现在写下的这些话。那么，当您偶然地拿起这本书，偶然地看到这段话，您是否会问自己：这样的偶然又会导致什么样的必然？

如果您依然决定继续这次的偶然之旅，还请您问自己一个问题：我在强迫自己学习 OpenStack 么？很希望您能回答不是，但希望与现实往往都有段不小的距离，因为很多时候，我们都是因为各种原因而强迫自己去喜欢的。或许，针对这个问题，最让人愉悦的回答是“说实话，我学习的热情从来都没有低落过。Just for Fun.”

其次，在您继续之前，面对 OpenStack 这样一个新生事物，让人最为惴惴不安的问题或许便是：我该如何更快更好的适应这个全新的世界？人工智能与机器学习领域里研究的一个很重要的问题是“为什么我们小时候有人牵一匹马告诉我们那是马，于是之后我们看到其他的马就知道那是马了？”。针对这个问题的一个结论是：我们头脑里形成了一个生物关系的拓扑，我们所认知的各种生物都会放进这个拓扑的结构里，而我们随着年纪不断成长的过程就是形成并完善各种各样或树形或环形等拓扑的过程，并以此来认知我们所面对的各种新事物。

由此可见，或许我们认知 OpenStack 最快也最为自然的方式就是努力在脑海里形成它的拓扑，并不断的进行细化。比如作为一个云计算的平台它包括了哪些功能分别对应哪些项目，各个项目又实现了哪些服务以及功能，这些功能又是以什么样的方式实现的，等等，对于我们感兴趣的项目或服务又可以更为细致的去勾勒它其中的脉络。就好似我们头脑里形成的有关一个城市的地图，它有哪些区，区里又有哪些标志建筑以及街道，对于我们熟悉的地方可以将它的周围进行放大细化，甚至于一个微不足道的角落。

而对于这个拓扑细化的过程能够起到有益辅助的是概念空间的勾勒。站在架构设计的角度，软件从需求进到架构出的全过程中，勾勒描绘概念空间是很重要的一个中间过程。这个阶段会形成所需要引入的各种新概念，比如操作系统中的进程、虚拟内存、系统调用等等，它们就类似一个拓扑中的标志建筑，而我们去认知研究这个软件的时候，描绘这个概念空间也就不可避免成为重中之重。

本书的组织形式

本书的内容组织正是为了尽一切能力帮助读者能够形成有关 OpenStack 以及各个重要项目与功能比较细致的拓扑。首先是前四章，这几章的内容希望能够帮助您对 OpenStack 有个全面的认识和了解，从而形成对 OpenStack 整体的拓扑。

第 1 章主要介绍了 OpenStack 的成长史以及它的体系结构和社区现状。

第 2 章详尽的介绍了 OpenStack 开发的基础流程以及如何去分析 OpenStack 的源码。

第 3 章介绍了 OpenStack 的底层基石——虚拟化技术。大多数 OpenStack 的使用者和开发者并不了解虚拟化的一些细节，有了这一章的介绍，我们能够对 OpenStack 有一个更好的认识。

第 4 章将 OpenStack 众多项目中所使用到的通用技术加以介绍，有了这一章，我们理解各个具体项目的设计与实现时，可以少去很多的阻碍。

然后第 5~14 章的内容对 OpenStack 主要组件及项目的实现进行介绍。按照认识的发展规律，通过前面几章的介绍我们已经对 OpenStack 有了全局的认识和了解，接下来就可以以兴趣或工作需要为导向，寻找一个组件或项目，对其实现进行深入的钻研和分析。这些章节的内容也是希望能够尽量帮助您形成对相应项目的比较细致的拓扑，并不求对所有实现细节的详尽分析。

第 5 章讨论计算组件也就是 Nova 项目。Nova 为我们实现了 OpenStack 这个虚拟机世界的抽象，控制着一个个虚拟机的状态变迁与生老病死，管理着他们的资源分配。

第 6 章讨论存储相关的四个项目：Swift, Cinder, Glance 以及 Ceph。他们共同为这个虚拟机世界的主体——虚拟机提供了安身之本，负责为每个虚拟机本身的镜像以及它所产生的各种数据提供一个家，尽量的去做到“居者有其屋”。

第 7 章讨论网络组件也就是 Neutron 项目。没有网络，任何虚拟机都将只是这个虚拟机世界中的孤岛，不知道自己生存的价值。

安全是每个软件无法回避的问题，第 8 章便针对安全问题进行讨论，包括 Keystone 项目以及可信计算池的相关内容。

第 9 章的内容有关计量与监控的项目 Ceilometer，计量与监控是公有云运营的一个重要环节。

第 10 章的内容与物理机管理有关，Ironic 项目被应用于 OpenStack 中的裸机管理和部署。

第 11 章介绍了 OpenStack 的控制面板。提供一个简洁方便、用户友好的控制界面给最终的用户和开发者对 OpenStack 尤为重要。

随着容器技术的发展，容器与云基础设施的结合受到越来越多的关注，第 12 章便讨论了 OpenStack 对容器的支持。

第 13 章的内容与部署有关，但是这里讨论的并不是如何部署的详细步骤与过程，而只是与部署有关的几个主要项目。

第 14 章介绍了几个新兴的项目，包括 Searchlight 与 Watcher 等。

感谢

作为英特尔的开源技术中心，参与 OpenStack 的开发与推广是再为自然不过的事情。除了为 OpenStack 的完善与稳定贡献更多的思考和代码，我们也希望能通过这本书让更多的人更快捷的融入 OpenStack 的大家庭。

如果没有 Imad Sousou（英特尔软件与服务事业部副总裁兼开源技术中心总经理）、Mauri Whalen（英特尔软件与服务事业部副总裁兼开源技术中心核心系统研发总监）、Hillarie Prestopine（英特尔软件与服务事业部副总裁兼开源技术中心云和网络系统研发总监）、David L Brown（英特尔开源技术中心云计算核心研发总监）、练丽萍（英特尔开源技术中心网络和存储研发总监）、Malini K Bhandaru（英特尔开源技术中心云计算主任工程师）、冯晓焰（英特尔开源技术中心中国安卓研发总监）、李少凡（英特尔开源技术中心虚拟化研发总监）、陈绪（英特尔开源技术中心中国云计算战略总监）的支持，这本书不可能完成，谨在此感谢他们对本书编写过程中的关怀与帮助。

也要感谢本书的编辑孙学瑛老师，从选题到最后的定稿，整个过程中，都给予我们无私的帮助和指导。

然后要感谢参与第一版与第二版各章内容编写的各位同事，他们是王庆、丁建峰、任桥伟、陆连浩、翟纲、徐贺杰、程盈心、李晓燕、臧锐、贺永立、郭瑞景、乔立勇、陈巍、杜永丰、杨林、张磊、冯少合、金运通、魏刚、田双太、汪亚雷、谭霖、辛晓慧，为了本书的顺利完成，他们付出了很多努力。他们不仅为英特尔开源技术中心做出了很多的贡献，而且长期活跃在中国的云计算技术生态系统中。

最后感谢所有对 OpenStack 抱有兴趣或从事 OpenStack 工作的人，没有你们的源码与大量技术资料，本书便会成为无源之水。

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与作者交流：**在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31199>



作者简介

英特尔开源技术中心 (Intel Open Source Technology Center -- OTC) 是英特尔公司内专职进行开源软件开发的团队, 负责在系统软件领域进行与英特尔®平台相关的技术开发和创新, 并在 Linux 操作系统内核, Android/Chrome 操作系统, 云计算和虚拟化技术, HTML5 Web Runtime, 图形驱动及多媒体软件以及系统软件的优化等方面积累了业界领先的技术能力。同时依托强大的技术团队, OTC 积极参与开源社区对开源软件的推广普及活动, 并对国内及国际上主流开源操作系统厂商提供有力地支持。



目 录

第 1 章 初识 OpenStack	1
1.1 从虚拟化到 OpenStack	1
1.1.1 虚拟化	1
1.1.2 云计算	2
1.1.3 OpenStack	4
1.2 OpenStack 基金会以及管理模式	7
1.2.1 董事会	8
1.2.2 技术委员会	9
1.2.3 用户委员会	10
1.3 OpenStack 体系结构	11
1.4 OpenStack 项目发展流程	18
1.4.1 新项目	18
1.4.2 孵化项目、集成项目和核心项目	19
1.4.3 大帐篷 (Big Tent)	20
1.5 OpenStack 社区	21
1.5.1 邮件列表	21
1.5.2 IRC 和项目例会	22
1.5.3 Summit 和 Meetup	23
1.5.4 其他社交平台	25
1.6 其他开源项目	25
1.7 OpenStack 的技术发展趋势	30
第 2 章 OpenStack 开发基础	33
2.1 相关开发资源	33
2.1.1 OpenStack 社区	33
2.1.2 OpenStack 文档	33
2.1.3 OpenStack 书籍	34
2.1.4 其他网络资源	35
2.2 OpenStack 开发的技术基础	35

2.3	部署开发环境	36
2.3.1	Git	37
2.3.2	Devstack	38
2.4	浏览 OpenStack 源代码	42
2.4.1	浏览代码的工具	43
2.4.2	分析源码如何入手	44
2.5	OpenStack 代码质量保证体系	48
2.5.1	编码规范	50
2.5.2	代码评审 Gerrit	53
2.5.3	单元测试 Tox	58
2.5.4	持续集成 Jenkins	61
2.6	如何贡献	66
2.6.1	文档	67
2.6.2	修补 bug	67
2.6.3	增加 feature	69
2.6.4	review	72
2.6.5	调试	73
第 3 章	虚拟化	75
3.1	概述	75
3.1.1	虚拟化实现方式	77
3.1.2	虚拟化现状和未来	79
3.2	高层管理工具	87
3.2.1	XenAPI	88
3.2.2	Libvirt	89
3.3	OpenStack 相关实现	98
3.3.1	Libvirt 驱动	98
3.3.2	XenAPI 驱动	100
第 4 章	OpenStack 通用技术	102
4.1	消息总线	102
4.2	SQLAlchemy 和数据库	107
4.3	RESTful API 和 WSGI	111
4.4	Eventlet	120

4.5	OpenStack 通用库 Oslo	121
4.5.1	Cliff.....	122
4.5.2	oslo.config.....	125
4.5.3	oslo.db.....	128
4.5.4	oslo.il8n.....	131
4.5.5	oslo.messaging.....	132
4.5.6	stevedore	139
4.5.7	TaskFlow.....	142
4.5.8	cookiecutter.....	149
4.5.9	oslo.policy.....	150
4.5.10	oslo.rootwrap	151
4.5.11	oslo.test	154
4.5.12	oslo.versionedobjects.....	156
第 5 章	计算.....	160
5.1	Nova 体系结构.....	161
5.2	Nova API	166
5.2.1	Nova v2.1 API.....	167
5.2.2	Nova API 实现.....	168
5.3	Rolling Upgrade.....	178
5.3.1	Rolling Upgrade 实现.....	179
5.4	Scheduler	186
5.4.1	调度器	187
5.4.2	Resource Tracker	191
5.4.3	调度流程	193
5.5	典型工作流程	195
5.5.1	创建虚拟机	195
5.5.2	冷迁移与 Resize.....	196
5.5.3	热迁移	198
5.5.4	挂起和恢复	200
5.5.5	Rebuild 和 Evacuate	200
第 6 章	存储.....	201
6.1	Swift.....	201

6.1.1	Swift 体系结构	201
6.1.2	Ring	209
6.1.3	Swift API	218
6.1.4	认证	226
6.1.5	对象管理与操作	228
6.1.6	数据一致性	231
6.2	Cinder	234
6.2.1	Cinder 体系结构	234
6.2.2	Cinder API	239
6.2.3	cinder-scheduler	241
6.2.4	cinder-volume	243
6.2.5	cinder-backup	248
6.3	Glance	249
6.3.1	Glance 体系结构	249
6.3.2	Glance API	252
6.4	Ceph	257
6.4.1	Ceph 体系结构	259
6.4.2	RADOS	261
6.4.3	Ceph 块设备	281
6.4.4	Ceph FS	285
6.4.5	Ceph 与 OpenStack	286
第 7 章 网络		289
7.1	Neutron 体系结构	289
7.1.1	Linux 虚拟网络	290
7.1.2	Neutron 网络抽象	294
7.1.3	Neutron 架构	295
7.1.4	Neutron 源码结构	297
7.2	Neutron API	299
7.2.1	neutron-server	300
7.3	ML2 Plugin	301
7.4	Port Binding 扩展	308
7.5	Open vSwitch Agent	317
7.6	Service Plugin	324
7.6.1	Firewall	325

7.6.2	LoadBalance	326
7.7	Neutron 热点话题	329
7.7.1	DVR	329
7.7.2	SDN	329
7.7.3	NFV/SRIOV	330
7.7.4	OVS 和 DPDK	333
第 8 章	安全	335
8.1	OpenStack 安全概述	335
8.2	Keystone	336
8.2.1	Keystone 体系结构	336
8.2.2	Keystone 启动过程	343
8.2.3	用户认证及令牌获取	346
8.2.4	签名证书生成	349
8.2.5	Keystone 高阶应用	352
8.3	可信计算池	355
8.3.1	体系结构	355
8.3.2	Intel TXT 与 TBoot	356
8.3.3	可信认证与 OpenAttestation 项目	358
8.3.4	TrustedFilter	362
8.3.5	部署	364
第 9 章	计量与监控	366
9.1	Ceilometer	367
9.1.1	体系结构	367
9.1.2	Pipeline	370
9.1.3	Polling Agent 与 Pollster 插件	372
9.1.4	Notification Agent 与 Notification Listeners 插件	373
9.1.5	Collector 与 Dispatcher 插件	373
9.1.6	Storage/DB	374
9.1.7	API Server	374
9.1.8	部署与使用	375
9.1.9	插件的开发	386
9.2	Aodh	396

9.2.1	体系结构	396
9.2.2	部署与使用	398
9.2.3	插件的开发	402
9.3	Gnocchi.....	408
9.3.1	体系结构	409
9.3.2	部署与使用	412
9.4	Panko	414
第 10 章 物理机管理.....		415
10.1	Ironic 体系结构	415
10.1.1	Ironic Driver.....	419
10.1.2	Ironic API.....	423
10.1.3	Ironic Conductor	424
10.1.4	Ironic-python-agent.....	425
10.1.5	ironic-inspector	426
10.2	Ironic 中的网络管理	426
10.2.1	物理交换机管理.....	426
10.2.2	多租户网络的支持.....	427
10.3	Ironic 节点的注册和启动	428
第 11 章 控制面板.....		432
11.1	Horizon 体系结构.....	432
11.1.1	Horizon 与 Django.....	432
11.1.2	Horizon 网站布局.....	435
11.1.3	Horzion 源码结构.....	437
11.2	Horizon 部署	439
11.3	页面渲染流程.....	441
第 12 章 容器.....		455
12.1	容器技术	455
12.1.1	容器的原理	455
12.1.2	常见的容器集群管理工具.....	456
12.2	容器与 OpenStack	460
12.2.1	nova-docker/heat-docker.....	461

12.2.2	Magnum	461
12.2.3	Murano	469
12.2.4	Kolla	472
12.2.5	Solum	472
12.2.6	Kuryr	474
12.2.7	容器技术与 OpenStack 的展望	476
第 13 章 部署		477
13.1	配置管理工具	478
13.2	OpenStack 部署项目	480
13.2.1	Bifrost	481
13.2.2	Kolla	483
13.2.3	TripleO	490
13.2.4	Fuel	493
第 14 章 新兴项目		495
14.1	Searchlight	495
14.1.1	Searchlight 体系结构	495
14.1.2	plugin 的开发	497
14.2	Watcher	502
14.2.1	Watcher 使用	503
14.2.2	Watcher 体系结构	505
14.2.3	strategy 的开发	507

初识 OpenStack

如果你尚未与 OpenStack 亲密接触过，那么希望这里的内容可以成为你初识 OpenStack 的见证。如果你已经是 OpenStack 达人，那么就选个安静的早晨，抑或下午，一起缅怀与 OpenStack 一起走过的青葱岁月吧。

1.1 从虚拟化到 OpenStack

至此落笔之际，OpenStack 已经成长了 4 年多，云计算被提出了 20 多年，虚拟化则发展了 50 多年，风雨颇多，感慨颇多，谨以这些许年来的点滴之事为献。

1.1.1 虚拟化

1. 1959 年

6 月，一个并不属于万物萌芽的月份。在 1959 年国际信息处理大会上，Christopher Strachey（克里斯托弗）亲手为虚拟化埋下了种子，他在名为《大型高速计算机中的时间共享》的报告中，提出了“虚拟化”的概念，从此拉开了虚拟化发展的帷幕。

2. 20 世纪 60 年代

虚拟化在这期间，由概念孕育到雏形，并得到了进一步的发展。1964 年，一种名为 CP-40 的新型操作系统首次实现了虚拟内存和虚拟机。随后，IBM 推出了 TSS（Time Sharing System，分时共享系统），允许多个用户远程共享同一高性能计算设备的使用时间，这也被认为是最为原始的虚拟化技术。

3. 20 世纪 70 年代

1972 年 IBM 发布了用于创建灵活大型主机的虚拟机技术，可以根据用户动态的应用需求来调整和支配资源，使昂贵的大型机资源得到尽可能的充分利用。虚拟化由此进入了大型机时代。

这一时期的 IBM System 370 系列通过一种叫虚拟机监控器（Virtual Machine Monitor，VMM）的程序在物理硬件之上生成许多可以运行独立操作系统软件的虚拟机实例，从而使虚拟机开始流行起来。

4. 20 世纪 80 年代与 90 年代

随着大规模集成电路的出现和个人电脑的普及，计算机硬件变得越来越便宜，当初为了共享昂贵的大型计算机资源而设计的虚拟化技术也由此渐渐无人问津，只是在一些高档的服务器中存在，虚拟化进入了“冷藏期”，遭遇了“成长的烦恼”。

这个阶段末期，随着 X86 技术的发展，X86 平台处理能力与日俱增。随着 Intel 于 1998 年推出专门针对服务器和工作站的 Xeon（至强）处理器，人们开始考虑将虚拟化技术引入用户面更为广泛的 X86 平台。

同时，VMware 公司于 1998 年成立，并随之于 1999 年在 X86 平台上推出了可以流畅运行的商业虚拟化软件，从此虚拟化技术终于走下大型机的神坛，进入了一个高速发展的阶段。

5. 21 世纪

VMware 的亮相，开启了虚拟化的 X86 时代，虚拟化的发展进入了一个爆发期。

2003 年，Xen 面世。同一年，微软因收购 Connectix 而获得虚拟化技术，进入桌面虚拟化领域，正式拉开了桌面虚拟化革命的序幕。

随后的 2004 年底，微软宣布了其 Virtual Server 2005 计划，被认为象征了“虚拟化正在从一个小市场向主流市场转变”。

2005 年，Intel 宣布其初步完成 Vanderpool 技术外部架构规范（EAS），并称该技术可帮助改进未来虚拟化解方案。并于同年 11 月，发布新的 Xeon MP 处理器系统 7000 系列，X86 平台历史上第一个硬件辅助虚拟化技术——VT（Vanderpool Technology）技术也随之诞生。

此后数年，AMD、Oracle、Redhat、Novell、Citrix、思科、惠普等先后进军虚拟化市场。

1.1.2 云计算

1983 年，Sun 提出“网络即是电脑”（“The Network is the Computer”），这被认为是云计算的雏形，而随后计算机技术的迅猛发展以及互联网行业的兴起，似乎都在向这个概念不断靠拢。

在这个不断靠拢的过程中，首先写上浓重一笔的是亚马逊。2006 年 3 月，亚马逊推出弹性计算云（Elastic Computing Cloud，EC2），按用户使用的资源进行收费，开启了云计算商业化的元年。

每一个时代的开始都有它自己的故事，而对于云计算时代，要从一篇文章说起。Steve Yegge 先后在亚马逊与 Google 公司工作，其于 2011 年在 Google+ 上和 Google 同事讨论有关平台的一些内容时，不小心把自己写的一篇辛辣调侃亚马逊与 Google 的文章向全世界公开，引起了剧烈的反应。

当然，事后，Steve 在其 Google+ 作了一些解释，大意是自己喝多了，又是凌晨，头脑不清，Google 对他很好，等等。但是这篇文章本身却堪称云计算架构的入门教材，中文翻译可见酷壳网（<http://coolshell.cn/>）上陈皓的一篇文章，这里着重提一下文中提到的 Jeff Bezos（亚

马逊创始人) 在 2002 年左右下的一份命令。

So one day Jeff Bezos issued a mandate. He's doing that all the time, of course, and people scramble like ants being pounded with a rubber mallet whenever it happens. But on one occasion -- back around 2002 I think, plus or minus a year -- he issued a mandate that was so out there, so huge and eye-bulgingly ponderous, that it made all of his other mandates look like unsolicited peer bonuses.

有一天, Jeff Bezos 下了一份命令。当然, 他总是这么干, 这些命令对人们的影响来说就像用橡皮槌敲击蚂蚁一样。这个命令大概是 2002 年, 误差应该是在 2002 年的前后 1 年内 —— 这个命令发布的范围非常广, 设想很大, 似乎都能让人的眼珠子掉出来, 就好像你突然收到公司给你的奖金一样让人惊讶。

His Big Mandate went something along these lines:

这份大命令大概有如下几个要点:

1) All teams will henceforth expose their data and functionality through service interfaces.

所有团队的程序模块都要以 Service Interface 方式将其数据与功能开放出来。

2) Teams must communicate with each other through these interfaces.

团队间的程序模块的信息通信, 都要通过这些接口。

3) There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

除此之外没有其他形式的通信方式: 不能直接链接程序、不能直接读取其他团队的数据库、不能使用共享内存、不能使用别人模块的后门, 等等, 唯一允许的通信方式只能调用 Service Interface。

4) It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols -- doesn't matter. Bezos doesn't care.

任何技术都可以使用。比如: HTTP、Corba、Pubsub、自定义的网络协议, 等等, Bezos 不管这些。

5) All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

所有的 Service Interface, 毫无例外, 都必须从骨子里到表面上设计成能对外开放的。也就是说, 团队必须做好规划与设计, 以便未来把接口开放给全世界的开发者, 没有任何例外。

6) Anyone who doesn't do this will be fired.

不这样的做的人会被炒鱿鱼。

7) Thank you; have a nice day!

谢谢，祝你有个愉快的一天！

在这份命令之后的几年，亚马逊内部转变成面向服务架构 (Service-Oriented Architecture, SOA), “一切以 Service 第一” 的系统架构成为该公司的企业文化。

You wouldn't really think that an online bookstore needs to be an extensible, programmable platform. Would you?

如果是你,你会想到要把一个在线卖书的网站设计成为一个有扩展性、可程序化的平台?你真的会这样想吗?

Well, the first big thing Bezos realized is that the infrastructure they'd built for selling and shipping books and sundry could be transformed an excellent repurposable computing platform. So now they have the Amazon Elastic Compute Cloud, and the Amazon Elastic MapReduce, and the Amazon Relational Database Service, and a whole passel' o' other services browsable at aws.amazon.com. These services host the backends for some pretty successful companies, reddit being my personal favorite of the bunch.

嗯,第一件 Bezos 领悟到的大事是,为了销售书籍和各种商品需要的基础架构,可以被转变成为绝佳计算平台 (Computing Platform)。所以,现在他们有了 Amazon Elastic Compute Cloud (亚马逊弹性运算云平台 EC2), Amazon Elastic MapReduce, Amazon Relational Database Service (亚马逊关系数据库服务),以及其他可到 AWS aws.amazon.com 查得到的一堆 Services。这些服务是某些相当成功的公司的后台架构,比如,我个人喜欢的 reddit 是这一堆成功公司的其中一个。

亚马逊之后, Google、IBM、雅虎、Intel、惠普等各大公司开始蜂拥进入云计算领域,并于 2010 年 7 月,美国国家航空航天局 (NASA) 与 Rackspace、Intel、AMD、戴尔等共同宣布 OpenStack 开放源码计划,由此开启属于 OpenStack 的时代。

1.1.3 OpenStack

1. 2010 年

2010 年 7 月, RackSpace 和美国国家航空航天局合作, 分别贡献出 RackSpace 云文件平台代码和 NASA Nebula 平台代码, 并以 Apache 许可证开源发布了 OpenStack。OpenStack 由此诞生。

OpenStack 第一版代号为 Austin, 以 RackSpace 所在的美国德州 Texas 首府命名, 计划每隔几个月发布一个全新版本, 并且以 26 个英文字母为首字母从 A 到 Z 顺序命名后面的版本代号。第一版 Austin 仅有 Swift 和 Nova 这两个项目, 分别来自 RaceSpace 云文件平台和 NASA Nebula 平台, 目的为云计算提供对象存储和计算平台。

2. 2011 年

2011 年 2 月, OpenStack 社区发布了 Bexar 版本。这是 OpenStack 的第二版, 此版本新增了一个项目 Glance 来提供镜像服务。

4 月, OpenStack 社区发布了更加稳定的 Cactus 版本, 但并没有新增任何项目。Ubuntu 的开发者很快地将 Bexar 版本吸收到 Ubuntu 11.04, 紧接着 Ubuntu 的母公司 Canonical 看到了其中的市场机会, 并宣布 Ubuntu 将全面支持 OpenStack。

9 月, OpenStack 发布了它的第四个版本 Diablo。OpenStack 诞生之初, 发行节奏很没有规律, 后面 OpenStack 社区逐步规范并计划发行节奏为每半年一次, 分别是当年的春秋两季。Diablo 是该节奏规范形成的第一个发行版本。

3. 2012 年

2012 年 4 月, OpenStack 又吸收了两个新的核心项目——用于用户界面操作的 Horizon 和认证的 Keystone, 并同时发行第五个版本 Essex。随后, Debian 7.0 集成了 Essex, 使得 Debian 用户可以直接使用 OpenStack 软件。Red Hat 也宣布集成 Essex 并发布 OpenStack 的第一个预览版。

8 月, Intel、新浪、中标软以及上海交通大学在北京联合成立“中国开源云联盟”(China Open Source Cloud League, COSCL), 旨在按照国际上 OpenStack 社区的工作方针, 整合中国 OpenStack 开发者和中国公司的研发资源, 深入参与 OpenStack 社区项目开发, 加大中国开发者和公司在国际 OpenStack 社区中的贡献力量。Intel 亚太研发有限公司总经理兼软件与服务事业部中国区总经理何京翔表示, “中国开源云联盟”将充分发挥 Intel 最新芯片的顶尖特性, 和合作伙伴合力打造高效的云端基础架构平台, 同时完全遵循开源规则, 积极向国际社区回馈代码。

9 月, OpenStack 社区将 Nova 项目中的网络模块和块存储模块剥离出来, 成立了两个新的核心项目, 分别是 Quantum 和 Cinder, 并发行了第六个版本 Folsom。

同一时期, OpenStack 基金会成立, 主席由 SUSE 开源部门总监兼 Linux 基金会董事 Alan Clark 担任。基金会最初拥有 24 名成员, 获得了 1000 万美元的赞助基金, RackSpace 的 Jonathan Bryce 担任常务董事。此后, OpenStack 项目纳入 OpenStack 基金会管理。9 月 8 日, Intel 成为 OpenStack 基金会金牌会员。

4. 2013 年

2013 年 4 月, OpenStack 发布了第 7 个版本 Grizzly, RedHat 也宣布在其商业发行版中对 OpenStack 提供全面的商业支持。

10 月, OpenStack 发布了第 8 个版本 Havana。在 Havana 中, 首次提出集成项目的概念, 并集成了两个新的项目, 分别是用于监控和计费的 Ceilometer 和用于编配 (Orchestration) 的 Heat。

5. 2014 年

4 月, OpenStack 发布了第 9 个版本 Icehouse, 并加入了一个新的项目 Trove 来提供数据库服务。

5 月, 在亚特兰大峰会上, OpenStack 发布了 Marketplace 项目计划。

7 月 19 日, IBM 工程师 Daisy 在北京海淀的车库咖啡发起了 OpenStack 第四个生日庆祝活动, 邀请了一些国内对社区有贡献的公司与个人。

10 月, 第十个版本 Juno 发布。

11 月, 在巴黎举行的峰会上, OpenStack 基金会白金会员 Nebula 退出, Intel 击败其他对手进入白金会员行列。

6. 2015 年

2015 年 4 月, Intel 与华为联合推动, 赶在 OpenStack 发布前夕, 成功在上海紫竹举办了第一届 OpenStack 黑客松活动。在短短 3 天时间里, 来自 3 家公司的 16 位开发者共修复了 29 个 Bug, 同月 OpenStack 发布了第十一个版本 Kilo。

5 月, OpenStack 在加拿大温哥华举办 Liberty 峰会, 并在该会上宣布了 OpenStack 互操作性测试认证, 即 OpenStack Powered 认证。当时首批 14 家厂商通过了该相关标准测试认证, 并被允许在其产品上贴有统一的 OpenStack Powered 标志。UnitedStack 是这些厂商中唯一的中国公司。

5 月, OpenStack 基金会、Intel、Red Hat 和计世传媒在北京成功举办 OpenStack 企业就绪论坛, 讨论 OpenStack 在企业私有云成熟性以及推动 OpenStack 在企业的大规模应用。

7 月, 来自 Intel 的王庆成功补选成为 OpenStack 基金会个人独立董事, 成为继程辉和杜玉杰之后, 第三位成功入选 OpenStack 基金会董事会的中国代表。19 日, 中国开源云联盟在北京发起并成功举办 OpenStack 五周年庆典。

8 月, 第二届 OpenStack 黑客松成功在西安举办。

10 月, Liberty 发布。Liberty 发布周期经历了开发模式的重大转变, 即取消了集成项目的概念, 启动了大帐篷 (Big Tent) 的发行模式。

11 月, OpenStack 东京峰会顺利举行。OpenStack 基金会同时宣布启动 OpenStack 管理员培训认证 (COA) 计划, 并发布一项新的工具 Project Navigator, 旨在让用户更好地理解项目成熟度等相关信息, 以帮助他们在如何使用软件方面作出明智的决策。

7. 2016 年

2016 年 2 月, 在社区会员投票中, 王庆以第三名的成绩成功继任 OpenStack 基金会个人独立董事。

3 月, 中国第三届 OpenStack 黑客松在成都举行, 这也是黑客松第一次推广到全世界 11 个城市同步举办, 包括纽约、悉尼、莫斯科、台北、班加罗尔、圣安东尼奥和美国湾区等地

区。

4月,在工信部信息化和软件服务业司的直接领导和关怀下,中国开源云联盟正式移交给中国电子技术标准化研究院。

同月,OpenStack迎来了第十三个发行版Mitaka的发布,并且OpenStack再次回到其诞生地美国奥斯汀举办Newton峰会。OpenStack基金会也对外正式宣布OpenStack管理员认证考试及全球首批COA认证培训机构。国内的九州云与其他十几家来自国外的企业一起成为首批COA认证培训机构。也就是在Newton峰会上,OpenStack基金会新增两家来自国内的黄金会员,即UnitedStack和EasyStack。

7月初,中国第四届黑客松在杭州举行,在落幕时参加者们投票选择出第五届黑客松的举办地为深圳。

来自Intel、华为和UnitedStack志愿者们经过3个月的筹备,7月14~15日成功在北京国家会议中心成功举办OpenStack中国日,这是OpenStack日系列活动第一次登陆中国大陆,当时吸引了2000多人参会,成为了OpenStack历史上人数规模最大的一次OpenStack日活动。

10月,Newton发行版正式发布。

1.2 OpenStack 基金会以及管理模式

2012年9月,OpenStack发行了第六个版本Folsom。也就是在这段时期,非营利组织OpenStack基金会成立,由SUSE的行业计划、新兴标准和开源部门总监兼Linux基金会董事Alan Clark担任基金会主席。

OpenStack基金会(<http://www.openstack.org/foundation/>)最初拥有24名成员,共获得了1000万美元的赞助基金,由RackSpace的Jonathan Bryce担任常务董事。OpenStack社区决定从此以后OpenStack项目都由OpenStack基金会管理。

OpenStack基金会的职责为推进OpenStack的开发、发布和作为云操作系统被采纳,并服务于来自全球的所有28000名个人会员。

OpenStack基金会的目标是为OpenStack开发者、用户和整个生态系统提供服务,并通过资源共享,推进OpenStack公有云和私有云的发展,辅助技术提供商在OpenStack中集成最新技术,帮助开发者开发出最好的云计算软件。

简单地说,OpenStack基金会是一非营利组织,由各公司资助会费,共同管理OpenStack项目,帮助推广OpenStack的开发、发行和应用。基金会会员有个人会员以及企业会员,个人会员是免费的、开放的,基金会鼓励个人会员参与技术贡献、代码贡献和社区建设。而针对公司会员,依据公司的决策及缴纳会费的多少,分为白金会员(Platinum Member)、黄金会员(Gold Member)、企业赞助会员(Corporate Sponsor)和支持组织(Supporting Organization)几种。

关于会员数量,OpenStack基金会只允许最多8家白金会员资格和24家黄金会员资格,

目前已有 AT&T、Canonical、惠普、IBM、Intel、Rackspace、红帽和 SUSE 这 8 家白金会员，以及 Aptira、CCAT 台湾云端运算产业协会、思科、戴尔、DreamHost、EMC、爱立信、富士通、日立、华为、inwinStack、Juniper、Mirantis、NEC、NetApp、Odin、赛门铁克和雅虎等黄金会员。

1.2.1 董事会

按照 OpenStack 基金会成立规则，所有 8 家白金会员和 24 家黄金会员中的 8 家是可以在董事会占有席位的，并由此具备各种事务的投票权。席位在基金会董事会里，是可以影响 OpenStack 发展和建设方向的，这也是企业对会员级别和董事会席位趋之若鹜的原因。

所有黄金会员需要通过投票竞争才能获得那 8 个黄金会员席位，投票由 24 个黄金会员们在一天内完成，不对外部社区公开。

最后，个人独立董事的 8 个席位，是由千万社区个人会员经过一周投票决定产生。所有这些 24 个席位构成了基金会董事会，如图 1-1 所示。



图 1-1 OpenStack 基金会董事会

董事会对 OpenStack 项目的管理、发展以及各项决策都有十分重要的决定权。比如，曾经所有被集成在 OpenStack 发行版中的项目都被称为核心项目，包括 Nova、Swift、Glance、Cinder、Neutron、Horizon 和 Keystone。但是在 2013 年，“核心”这个词变成了 OpenStack 基金会董事会能在 OpenStack 发行版里对某个项目进行标签的特有名词，“核心”的使用也就被限制了，于是此后被集成的项目被称为集成项目，再后来，子项目越来越多，OpenStack 允许子项目自己决定自己的发布，经过一些流程审核通过且被选中的子项目，我们称之为大帐篷项目，这一系列决策都来自董事会。

一般来说，基金会会成立各种工作组（Working Group 或 WG），有计划、有目标地做一些推动 OpenStack 发展的事情。比如 2014 年亚特兰大峰会上，Intel 代表提出发起建立企业就

绪工作组（Win the Enterprise WG 或 Enterprise WG），其目的是为了推动 OpenStack 从公有云向私有云转化，为推动 OpenStack 企业就绪进行相应的工作。后来因为既要考虑企业就绪，又要考虑电信就绪等市场，干脆就成立了一个产品工作组（Product WG），显得更为专业。这个工作组的工作内容包括定义产品工作组的目标和工作方式、定义各时间段的 Roadmap、交付时间表以及工作流程、定义用户委员会的介入方式，以及介绍 PTL 如何在工作组里收集反馈并把反馈转化成将来开发的功能，等等。

产品工作组定义有 3 个目标：

- 放大来自市场/用户/运维在 OpenStack 设计和开发工作流中的声音，即 OpenStack 设计和开发应该有效地尊重并考虑来自于市场/用户/运维的实际需求。
- 简化跨项目功能的定义、实现和跟踪。
- 发布 OpenStack 的 Roadmap 以帮助运维/用户/其他人可以事先规划好自己的部署。

董事们在董事会会议期间，需要听取各方报告，有时还需要投票批准相关文件。另外，每次在峰会的首日，基金会也会举行董事会，听取来自 Jonathan Bryce 的 OpenStack 基金会工作人员例行报告，了解 OpenStack 运维的健康状况。报告内容不仅包括工作总结，还包括峰会准备情况，以及财务收支情况等。

1.2.2 技术委员会

OpenStack 基金会在成立之初就设立了专门的技术委员会，来指导 OpenStack 技术相关的工作，如图 1-2 所示。对于技术问题讨论、某项技术决策和未来技术展望，技术委员会负责提供指导性建议和意见。除了技术指导之外，技术委员会还要确保 OpenStack 项目的公开性、透明性、普遍性、融合性和高质量。

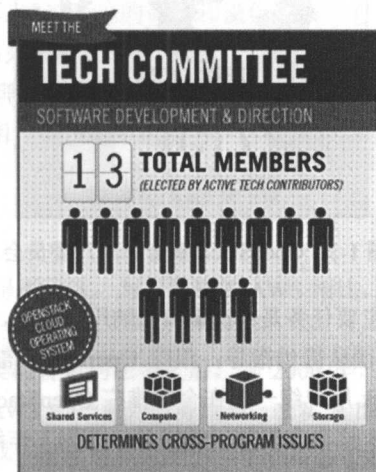


图 1-2 OpenStack 基金会技术委员会

一般情况下，OpenStack 技术委员会由 13 位成员组成，这 13 位成员完全是 OpenStack 社区中有过代码贡献的开发者投票选举出来的，通常任职 6 个月后需要重选。有趣的是，其中的 6 位成员是在每年秋天选举产生，另外 7 位是在每年春季选举产生，通过时间错开保持了该委员会成员的稳定性和延续性。技术委员会成员候选人的唯一条件是该候选人必须是 OpenStack 基金会的个人成员，除此之外，不作其他要求。而且，技术委员会成员也可以同时在 OpenStack 基金会其他部门兼任职位。

技术委员会在选举产生之后，会提名 13 位中某一位担任技术委员会主席。如果有多位候选人被提名，则采取投票的方式和少数服从多数的原则决定。除非有特殊情况，例如法律规定禁止外，OpenStack 基金会董事会有权利也会相应地批准最终技术委员会主席的任命。技术委员会主席负责组织定期会议，并及时与基金会董事会和整个社区沟通。

1.2.3 用户委员会

随着越来越多的用户在生产环境中使用 OpenStack，以及 OpenStack 生态圈里越来越多的合作伙伴在云中支持 OpenStack，社区指导用户使用和产品发展的使命就变得越来越重要。鉴于此，OpenStack 用户委员会应运而生，如图 1-3 所示。

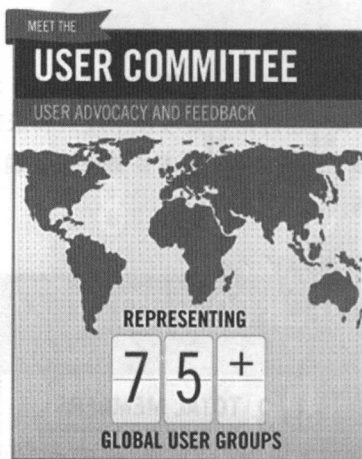


图 1-3 OpenStack 基金会用户委员会

OpenStack 用户委员会的主要任务是：收集和归纳用户需求，并向董事会和技术委员会报告；以用户反馈的方式向开发团队提供指导；跟踪 OpenStack 部署和使用，并在用户中分享经验和案例；与各地 OpenStack 用户组一起在全球推广 OpenStack。

OpenStack 用户委员会由 3 位个人领导，并指导一系列工作组的工作。

1.3 OpenStack 体系结构

“云计算”中所谓的“云”可以简单被理解为任何可以通过互联网访问的服务，那么根据所提供服务的类型，云计算就有 3 种落地方式。

- **IaaS（基础架构即服务）**：通过互联网提供“基础的计算资源”，包括处理能力、存储空间、网络等，用户能从中申请到硬件或虚拟硬件，包括裸机（Bare Metal）或虚拟机，然后在上面安装操作系统或其他应用程序。
- **PaaS（平台即服务）**：把计算环境、开发环境等平台作为一种服务通过互联网提供给用户。用户能从中申请到一个安装了操作系统以及支撑应用程序运行所需要的运行库等软件的物理机或虚拟机，然后在上面安装其他应用程序，但不能修改已经预装好的操作系统和运行环境。
- **SaaS（软件即服务）**：通过互联网，为用户提供软件及应用程序的一种服务方式。应用软件安装在厂商或者服务供应商那里，用户可以通过网络以租赁的方式来使用这些软件，而不是购买。比较常见的模式是提供一组账号密码。

PaaS 和 SaaS 并不一定需要底层有虚拟化技术的支持，但 IaaS 一般都是建立在虚拟化技术基础之上的。OpenStack 以及它一直在跟随的榜样 AWS 都是属于 IaaS 的范畴。

本质上，IaaS 系统其实就是一个用户层的软件系统，它包含多个服务和应用程序，这些服务或程序被部署到多台被管理的物理主机上，这些物理主机通过网络相连从而形成一个大的分布式系统。IaaS 系统要解决的问题就是如何自动管理这些物理主机上虚拟出来的虚拟机，包括虚拟机的创建、迁移、关闭，虚拟存储的创建和维护，虚拟网络的管理，还包括监控计费、负载均衡、高可用性、安全等。这里不提供任何虚拟化服务的裸机（Bare Metal）也被视为虚拟机的一种特例，对它的管理也属于虚拟机管理的范畴。

在单个主机上，这些都可以通过简单的命令和操作完成，有些问题，比如高可用性或负载均衡等根本不存在。但是，如果在大规模网络上或数据中心里，将有成千上万台物理主机，仅仅靠运维人员来完成这些管理任务是不现实的，这时候就需要软件系统来自动辅助运维人员管理和维护系统的运行，给用户提供虚拟机服务。这就是 IaaS 系统产生的初衷，也是 AWS 和 OpenStack，以及其他 IaaS 产品和开源项目要实现的功能。

1. OpenStack 与 AWS

无论是否情愿，我们都不得不承认，与亚马逊的 AWS 相比，OpenStack 只是处于一个跟随者的位置。

Bezos 颁布那份充满系统架构智慧的法令之后，到 2006 年，亚马逊推出了 AWS 产品，正式开启了云计算的新纪元。AWS 由一系列服务组成，去实现 IaaS 系统所需要的功能。如图 1-4 所示，AWS 架构由 5 层组成，自下而上分别是 AWS 全球基础架构、基础服务、应用平台服务、管理和用户应用程序。

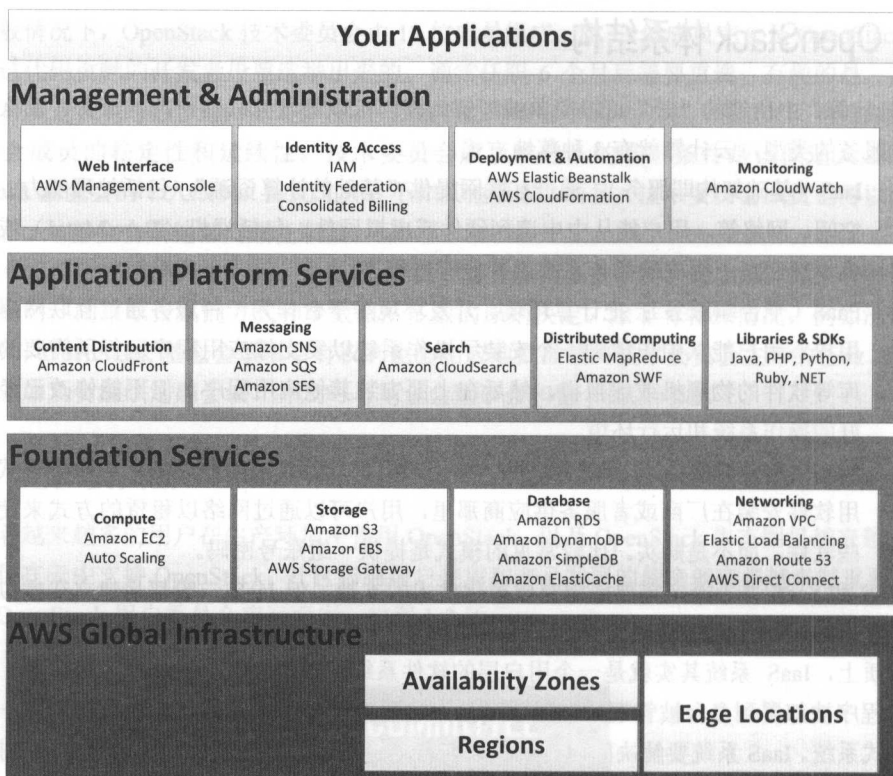


图 1-4 AWS 架构

而就服务类型本身而言，AWS 主要提供 6 类服务：计算和网络、存储和内容分发、数据库、分析、部署管理和应用服务，如图 1-5 所示。

计算和网络服务中涵盖了负责虚拟机调度和管理的弹性计算云 EC2、用于计算资源自动扩容缩容的 Auto Scaling 服务、负载均衡服务 ELB、虚拟桌面管理服务 WorkSpaces、保证企业在公有云上搭建安全私有云的虚拟私有云服务 VPC、高可靠且可扩展的域名系统 Web 服务 Route 53 以及为企业定制的专属网络连接 Direct Connect 等。

在存储和内容转发服务中涵盖了简单存储服务 S3、Amazon Glacier、块存储 EBS、AWS 存储网关、AWS 导入导出以及 Amazon 云前端。其中 S3 提供 AWS 永久存储服务，而 EBS 提供的是块存储服务。

在数据库服务里，包括关系数据库服务 RDS、NoSQL 数据库服务 DynamoDB、缓存和数据仓库服务 RedShift。

在分析服务里，AWS 包括用于大数据的弹性 MapReduce (EMR)、用于大规模实时流数据处理的 Kinesis 和数据管道。

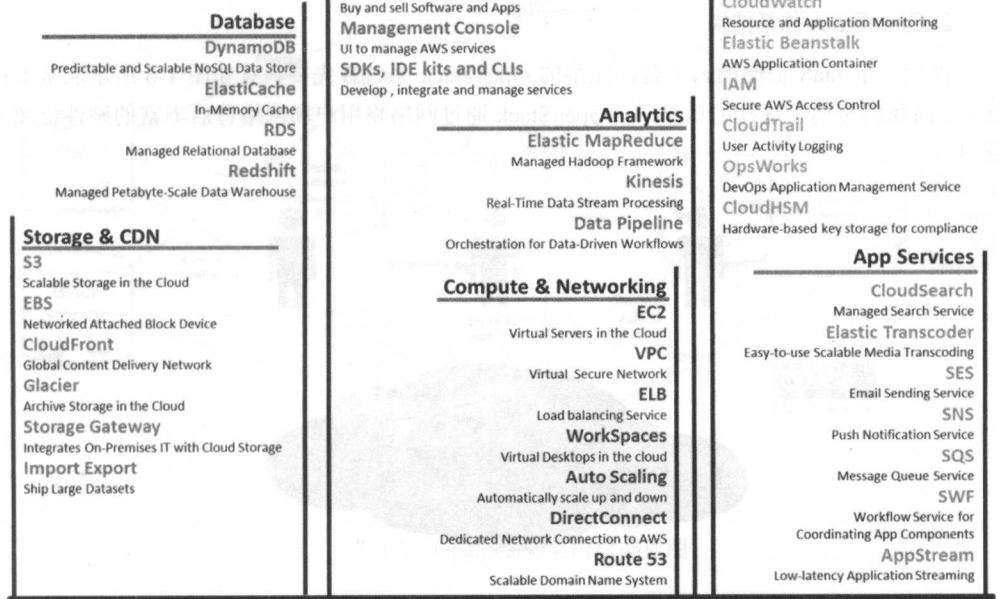


图 1-5 AWS 的服务模块

在部署管理服务里,AWS 包括验证访问管理 IAM、日志管理 CloudTrail、监控 CloudWatch、用于轻松部署 Web 应用和服务的 Beanstalk、建立和管理 AWS 资源的 CloudInformation、为运维人员配备的应用管理服务 OpsWorks 和安全服务 CloudHSM。

在应用服务里,AWS 包括应用程序流(AppStream)、简单队列服务(SQS)、简单消息服务(SNS)、简单工作流服务(SWF)、简单邮件服务(SES)、用于云搜索的 CloudSearch,以及用于流媒体转码的 Transcoder。

如上所述,AWS 的功能十分强大,而且目前还在不断发展之中,OpenStack 从诞生之初就一直向 AWS 模仿和学习,同时,OpenStack 也提供开放接口去兼容各种 AWS 服务。

比如,AWS 中最为核心的 EC2 模块,负责计算资源的管理,以及虚拟机的调度和管理,在 OpenStack 中对应的就是 Nova 项目;AWS 中的简单存储服务 S3,在 OpenStack 中有 Swift 项目与其功能相近;AWS 中的块存储模块 EBS,对应 OpenStack 的 Cinder 项目;AWS 的验证访问管理服务 IAM,对应 OpenStack 的 Keystone 项目;AWS 的监控服务 CloudWatch,对应 OpenStack 的 Ceilometer 项目;AWS 有 CloudInformation,OpenStack 则有 Heat 项目;AWS 支持关系数据库 RDS 和 NoSQL 数据库 DynamoDB,OpenStack 也支持 MySQL、postgre 和 NoSQL 数据库 MongoDB。

目前来看,OpenStack 远没有 AWS 完善,但是相比来说,OpenStack 是开源项目,并没

有版权费用，这一点吸引了众多的企业 IT 人员、开源爱好者和开发人员、学术界人士、云服务提供商、运维人员等加入到 OpenStack 社区，并贡献代码和分享经验。

2. OpenStack 体系结构

作为一个 IaaS 范畴的云平台，完整的 OpenStack 系统首先要具有如图 1-6 所示的基本视图，它向我们传递了这样的信息——OpenStack 通过网络将用户和网络背后丰富的硬件资源分离开。

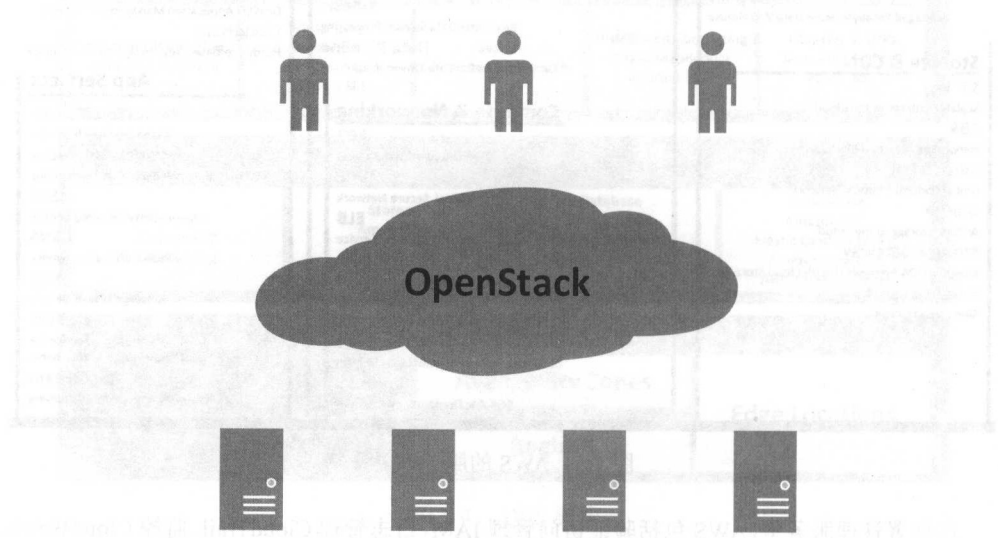


图 1-6 OpenStack 基本视图

OpenStack 一方面负责与运行在物理节点上的 Hypervisor 进行交互，实现对各种硬件资源的管理与控制；另一方面为用户提供一个满足要求的虚拟机。

至于 OpenStack 内部，作为 AWS 的一个跟随着，它的体系结构里不可避免地体现着前面所介绍的 AWS 各个组件的痕迹。图 1-7 所示为 OpenStack 架构标准视图。

图 1-7 涵盖了 OpenStack 曾经的 7 个核心组件，分别是计算 (Compute)、对象存储 (Object Storage)、认证 (Identity)、用户界面 (Dashboard)、块存储 (Block Storage)、网络 (Network) 和镜像服务 (Image)。这 7 个核心组件除用户界面外，其余 6 个仍是目前的核心组件。每个组件都是多个服务的集合，一个服务意味着运行着的一个进程，根据部署 Openstack 的规模，决定了你是选择将所有服务运行在同一个机器上还是多个机器上。

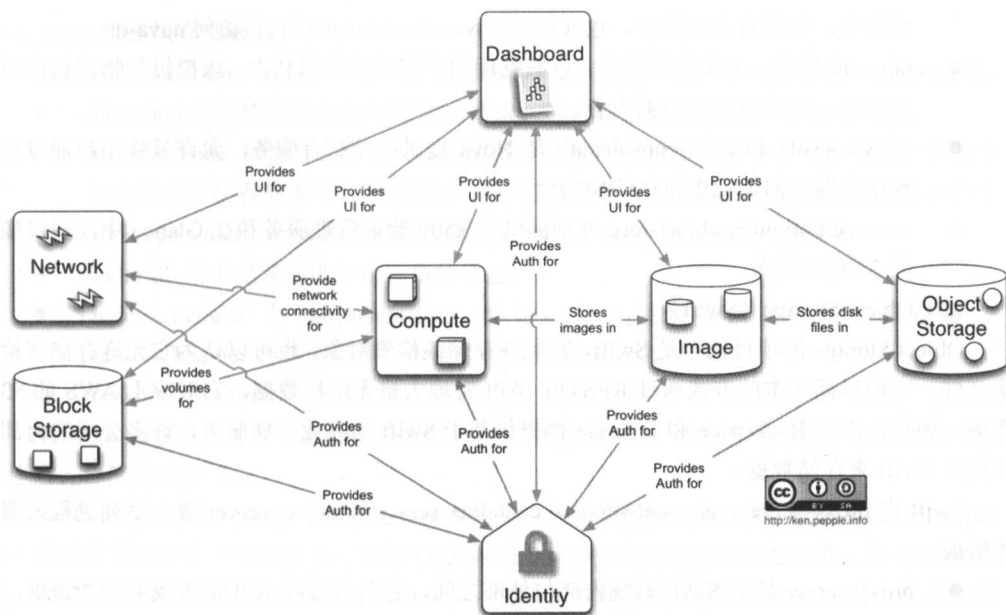


图 1-7 OpenStack 架构标准视图

(1) Compute (Nova)

Compute 的项目代号是 Nova，它根据需求提供虚拟机服务，比如创建虚拟机或对虚拟机做热迁移等。从概念上看它对应于 AWS 的 EC2 服务，而且它实现了对 EC2 API 兼容。如今，Rackspace 和惠普提供商业计算服务正是建立在 Nova 之上的，NASA 内部使用的也是 Nova。

Nova 组件包括 nova-api、nova-compute、nova-scheduler、nova-conductor、nova-db、nova-network、nova-console、nova-consoleauth、nova-cert 和 nova-objectstore 等。

- nova-api 是 Nova 对外提供服务的窗口，它接收并响应来自用户的 Compute API 调用。nova-api 同时兼容亚马逊 AWS EC2 API，也提供一套管理员操作相关的管理 API。
- nova-compute 是安装到每个物理主机上的服务进程，这个服务接收请求之后执行一批与虚拟机相关的操作，这些操作需要调用底层的 Hypervisor API 完成，比如支持 XenServer/XCP 的 XenAPI、支持 KVM 和 QEMU 的 libvirt 或者支持 VMware 的 VMwareAPI 等。
- nova-scheduler 用于接收创建虚拟机的请求，并决定在哪台物理主机上启动该虚拟机的调度器。
- nova-conductor 是处于 nova-compute 与 nova-db 之间的一个组件。nova-conductor 建立的初衷是基于安全的考虑而避免 nova-compute 直接访问 nova-db。也就是说，nova-compute 对 nova-db 的访问请求，比如让 nova-compute 查询一台虚拟机的状态，或更新一条记录，都由 nova-conductor 代为转交。而对于 nova-scheduler 对 nova-db

的请求，却没有这种顾虑，也就是说 nova-scheduler 可以直接访问 nova-db。

- nova-db 包含一大堆数据库表，该数据库用于记录虚拟机状态、虚拟机与物理机的对应关系、租户信息等数据内容。
- nova-console 和 nova-consoleauth 是 Nova 提供的控制台服务，允许最终用户通过代理服务器访问他们虚拟机的控制台。
- nova-cert 和 nova-objectstore 分别提供了 x509 验证管理服务和在 Glance 中注册镜像的 S3 接口服务。

(2) Object Storage (Swift)

Object Storage 的项目代号是 Swift，它允许存储或检索对象，也可以认为它允许存储或检索文件，它能以低成本的方式通过 RESTful API 管理大量无结构数据。它对应于 AWS 的 S3 服务。如今，KT、Rackspace 和 Internap 都提供基于 Swift 的商业存储服务，许多公司的内部也使用 Swift 来存储数据。

Swift 由 proxy-server、account-server、container-server 和 object-server 等一系列进程或服务组成。

- proxy-server 处于 Swift 系统内部与外部之间，它负责接收 API 请求或 HTTP 请求，这些请求包括上传文件、修改元数据、创建容器等。有些时候，为了提高系统性能，proxy-server 也会与 memcached 在一起部署。
- account-server 仅用于账号管理。
- container-server 用于管理容器或文件夹的映射关系。
- object-server 用于管理在存储节点上的实际对象，比如文件等。

除此之外，还有一些定期执行的进程，比如 replication、auditor、updater 和 reaper 等。

(3) Identity (Keystone)

Identity 的项目代号是 Keystone，为所有 OpenStack 服务提供身份验证和授权，跟踪用户以及他们的权限，提供一个可用服务以及 API 的列表。

Keystone 构成组件并不复杂，只包括接收前台请求的 Keystone API 和后台的 keystone-db。

(4) Dashboard (Horizon)

Dashboard 的项目代号是 Horizon，它为所有 OpenStack 的服务提供一个模块化的基于 Django 的界面。通过这个界面，不论是最终用户还是运维人员都可以完成大多数的操作，比如启动虚拟机、分配 IP 地址、动态迁移等。

(5) Block Storage (Cinder)

Block Storage 的项目代号是 Cinder，提供块存储服务。Cinder 最早是由 nova-volume 演化而来的，当时由于 Nova 已经变得非常庞大并拥有众多功能，也由于 volume 服务的需求会进一步增加 nova-volume 的复杂度，比如，增加 volume 调度，允许多个 volume driver 同时工作，同时考虑到需要 nova-volume 跟其他 OpenStack 项目交互；将 Glance 中的镜像模板转成可启动的 volume，所以 OpenStack 新成立了一个项目 Cinder 来扩展 nova-volume 的功能。Cinder

对应 AWS EBS 块存储服务。

Cinder 由 cinder-api、cinder-volume、cinder-db、volumeprovider 和 cinder-scheduler 组成。

- cinder-api 用于接收来自外部的 API 请求，并把请求交给 cinder-volume 执行。
- cinder-volume 负责与底层的块存储服务打交道，它响应读和写块设备请求，并把这个请求交给块存储服务，底层的不同存储服务提供商都通过 driver 的方式实现了 volumeprovider，所以具体的读写请求交给底层 volumeprovider 来完成。
- cinder-db 用于记录和维护块设备的信息。
- cinder-scheduler 与 nova-scheduler 类似，由于底层提供存储的节点很多，cinder-scheduler 会试图寻找一个最佳的节点创建 volume。

(6) Network (Neutron)

Network 的项目代号是 Neutron，用于提供网络连接服务，允许用户创建自己的虚拟网络并连接各种网络设备接口。

Neutron 通过插件(plugin)的方式对众多的网络设备提供商进行支持，比如 Cisco、Juniper 等，同时也支持很多流行的技术，比如 Openvswitch、OpenDaylight 和 SDN 等。与 Cinder 类似，Neutron 也来源于 Nova，即 nova-network，它最初的项目代号是 Quantum，但由于商标版权冲突问题，后来经过提名投票评选更名为 Neutron。

Neutron 包含的组件有 neutron-server、neutron-agent、network-provider、neutron-plugin，以及用于保存网络配置等相关信息的 neutron-db 等。

- neutron-server 用于接收来自外部的 API 请求，并将该请求交给合适的 neutron 插件处理。

在 neutron 当中，有众多的插件和代理 (Agent)，它们负责插拔端口、建立网络和子网、提供 IP 地址等。插件从功能上来说用于存储当前逻辑网络的配置信息，判断和存储逻辑网络与物理网络之间的对应关系，以及与一种或多种交换机通信来实现这种对应关系，它需要访问 neutron-db。

而实现这种对应关系，一般需要通过物理机上的代理来完成，代理可以分为 plugin agent、DHCP agent 和 L3 agent。虚拟网络上数据包的处理都是由 plugin agent 完成的，一般选择了什么插件，就需要选择相应的 plugin agent，plugin agent 会调用相应的 network-provider 完成与该网络设备对应的功能；DHCP agent 指的是为租户网络提供 DHCP 服务，每个插件都是使用这一个代理；L3 agent 指的是为虚拟机访问外部网络提供 3 层转发服务。

(7) Image Service

Image Service 的项目代号是 Glance，它是 OpenStack 的镜像服务组件，相对于其他组件来说，Glance 功能比较单一代码量也比较少，而且由于新功能的开发数量越来越少，近來社区的活跃度也没有其他组件那么高，但它仍是 OpenStack 核心项目之一。

Glance 主要提供一个虚拟机镜像的存储、查询和检索服务，通过提供一个虚拟磁盘映像的目录和存储库，为 Nova 的虚拟机提供镜像服务。它与 AWS 中的 Amazon AMI catalog 功能

相似。

Glance 由 glance-api、glance-registry 和 glance-db 等组件组成。

- glance-api 用于接收来自外部的 API 镜像请求, 这些请求包括镜像发现、获取及存储。
- glance-registry 用于存储、处理和获取镜像元数据。对于 v2 版本的 API, glance-registry 的功能已经被整合到 glance-api 中。
- glance-db 里存储的就是元数据。

现在以创建虚拟机为例, 来介绍这些核心组件是如何相互配合完成工作的。用户首先接触到的是界面, 即 Horizon。通过 Horizon 上的简单界面操作, 一个创建虚拟机的请求被发送到 OpenStack 系统后端。

既然要启动一个虚拟机, 就必须指定虚拟机操作系统是什么类型, 同时下载启动镜像以供虚拟机启动使用。这件事情就是由 Glance 来完成的, 而此时 Glance 所管理的镜像有可能存储在 Swift 上, 所以需要与 Swift 交互得到需要的镜像文件。

在创建虚拟机的时候, 自然而然地需要 Cinder 提供块服务和 Neutron 提供网络服务, 以便该虚拟机有 volume 可以使用, 能被分配到 IP 地址与外界网络连接, 而且之后该虚拟机资源的访问要经过 Keystone 的认证之后才可以继续。至此, OpenStack 的所有核心组件都参与了创建虚拟机的操作。

1.4 OpenStack 项目发展流程

经过 6 年的发展, OpenStack 已经从最初的只有 Nova 和 Swift 两个项目发展到目前有形形色色上百个项目, 但是 OpenStack 每一个版本的发布, 仅仅只是包含其中很少量的一些, 那么这里的问题就是一个项目从最初的创建到最终被包含在 OpenStack 发布版之中, 需要经历什么样的阶段?

1.4.1 新项目

互联网时代最缺的是什么? 毫无疑问是 Idea。同样 OpenStack 里一个新项目的源泉也是新的 Idea。这个 Idea 的产生既可能来自 AWS 里的功能, 也可能来自于 OpenStack 使用者的需求。当这个 Idea 逐渐成熟而且工作量足够大, 以致无法在现有的某个 OpenStack 项目中承载时, 就有必要成立一个独立的新项目去开发。

项目的发起者可以是一个人, 但更有可能的是一群人。他们会发动开源社区, 推广这个新项目并吸引一批开发者共同开发。这些开发者形成的团队会在 OpenStack 邮件列表上讨论问题, 并定期举办日常例会。

新项目成立早期, 如果还没有 PTL (Program Technical Lead, 技术领头人), 团队内部会选举指派一个领头人带领整个团队的开发, 以及主持每期例会。

由于该项目是开源的, 就会源源不断地有新的开发者加入到开发团队当中, 同时, 也会

有人去审视并吸收类似的开源项目，以避免重复工作。逐步地，项目渐渐成熟，形成自己的目标、计划和代码库。为了方便起见，项目发起者们一般会先将项目存放在 `stackforge` 目录上。

对于最初项目的版权，最好是 Apache 2.0，这样就与 OpenStack 保持一致。当有一天新项目被集成到 OpenStack 发布版中时，也就不需要重新定义和处理版权问题。

值得一提的是，当一个项目还是新项目的时候，它是在 OpenStack 之外进行开发的，这是该项目必须经历的一个阶段，但是项目发起者们却可以利用任何 OpenStack 正在使用的工具去管理该项目，例如使用 Launchpad 工具作为跟踪 bug 和 Blueprint 的工具。

经过若干月或若干年，一旦项目发起者认为该项目足够成熟了，他们就可以向 OpenStack 技术委员会提出 OpenStack 孵化请求，等待成为 OpenStack 孵化项目的批准。

1.4.2 孵化项目、集成项目和核心项目

1. 孵化项目

在一个项目被集成到 OpenStack 发布版之前，成为孵化项目是必经阶段。在这个阶段里，项目开发人员需要了解 OpenStack 的发布节奏、发布流程及要成为集成项目还有哪些工作需要完成等内容。同时，也可以尽量寻求与其他项目合作或合并的机会。一般来说，这个阶段至少需要持续两个开发周期。

在孵化期间，孵化项目都会被移植到 `openstack` 命名空间和目录中。在一个开发周期结束时，OpenStack 技术委员会会对孵化项目做一个考核，理论上只有经历了两个开发周期的孵化项目才能被选为考核目标。

2. 集成项目

考核的结果如果被证明是足够成熟并且已经准备好被集成到 OpenStack 发布版当中的，就会被选择从孵化期“毕业”成为 OpenStack 集成项目。在下一个开发周期里，该项目就正式成为集成项目，成为 OpenStack 正式家族成员之一。

比如，监控计费项目 Ceilometer 成立于 2012 年，当时最初的想法是能提供一套基础架构用于收集来自于各种 OpenStack 项目中的数据，为运营商计费提供参考依据。Ceilometer 在 2012 年 Grizzly 开发周期里成为孵化项目，并在 2013 年 4 月 Havana 开发周期里，正式地被批准成为 OpenStack 的集成项目。

3. 核心项目

核心项目的含义在 2013 年有所改变，那时 OpenStack 项目的管理刚刚被转交给 OpenStack 基金会。在此之前，所有被集成在 OpenStack 发布版中的项目都被称之为核心项目，包括 Nova、Swift、Glance、Cinder、Neutron、Horizon 和 Keystone。

此后，“核心”这个词变成了 OpenStack 基金会在 OpenStack 发布版里对某个项目进行标

签的特有名词,“核心”的使用也就被限制了。可以这么说,核心项目是集成项目的一部分,是它的子集。当 OpenStack 基金会的董事会认为某一个集成项目能达到某些要求时,可以为该集成项目贴上“核心”这个标签。

在 2013 年之后,所有从孵化期毕业被集成在 OpenStack 发布版里的项目都统一称作集成项目,比如 Ceilometer、Heat 和 Trove 都是集成项目,但针对之前的那 7 个核心项目,我们仍称它们为核心项目。

不论是核心项目还是集成项目,都代表着该项目已经成为 OpenStack 的正式家庭成员,在每半年 OpenStack 版本的发布里,都会包含该项目。这也意味着对核心项目和集成项目有着更高的要求,不论是新功能开发流程以及代码稳定程度,还是项目的开发周期,它们都必须与 OpenStack 的节奏保持高度一致。

OpenStack 的项目“孵化集成”发展模式流程,如图 1-8 所示。

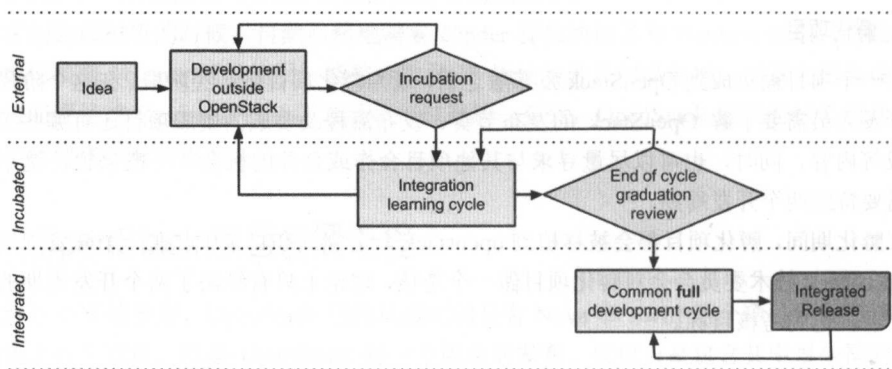


图 1-8 OpenStack 项目“孵化集成”发展模式流程

1.4.3 大帐篷 (Big Tent)

但是,从 2015 年 Liberty 发布版开始,OpenStack 废弃了原来的孵化集成模式,进入了一个全新的发行模式,即引入了所谓的“大帐篷”一词,使得 OpenStack 发行变得更加松散和自由。

在之前的孵化集成模式里,新项目先要被孵化,然后成熟到一定程度的时候,经过申请和投票决定是否把它放到集成项目里,而且它的发布版必须与 OpenStack 发布同步。

这样,集成项目丧失了自由度和灵活性。随着云计算技术的快速发展,用户需求的快速增长和动态变化,大量新项目在希望成为 OpenStack 项目时就遇到了瓶颈。为了满足市场动态变化,大帐篷模式应运而生。OpenStack 基金会用文档定义了 OpenStack 流程以及其开源社区的工作方式,只要该项目符合这些流程,经过审核批准,就可以成为大帐篷的一员,进而成为 OpenStack 项目,而且该项目的发行步骤也可以根据自身的特殊要求做小范围调整,没

有必要与 OpenStack 发布版完全一致。比如，如果没有大帐篷，也许 Magnum 项目就不会这么容易地成为 OpenStack 项目。

一个大帐篷项目只负责在某个空间里解决某一特定的问题，但由于成为大帐篷项目变得更容易，这种机制也更容易地引入了竞争，也就是说，很有可能在大帐篷里有多个项目是解决同一或类似问题的，但对于用户来说这反而是件好事情，用户的选择范围更加广了。

在大帐篷里，OpenStack 仍然保留了 6 个核心项目，即 Nova、Cinder、Swift、Neutron、Keystone 和 Glance，只有原核心项目 Horizon 现在被放在非核心项目里。而以前的集成项目都无一例外地放在大帐篷的非核心项目中。

诚然，大帐篷也有一定的缺点。比如，某些项目极可能是来自同一家公司开发者主导开发的，那么该公司会对这些项目具有绝对的话语权，使得这些项目慢慢地朝着垄断和封闭的方向发展。另外，大帐篷使新项目变成 OpenStack 项目的门槛降低，会使得 OpenStack 涌现大批项目，有些项目甚至不知道是做什么、解决什么问题的，而且项目质量也难以掌控，这增加了基金会的管理难度。

1.5 OpenStack 社区

对于一个开源软件来说，与之相关的开源社区无疑是最为重要的，它的活跃程度决定了这个开源软件的发展前景与想象空间。

与 Linux 社区一样，OpenStack 也有一个紧密团结了众多使用者和开发者的 OpenStack 社区，在这个属于 OpenStack 的世界里，开发人员可以讨论什么样的设计是最优的，运维人员也可以提出使用 OpenStack 的反馈意见和需求建议。

此外，OpenStack Summit 的举行，下一版 OpenStack 应该取什么样的名字，也都是在社区里完成提名和投票的。OpenStack 社区官方主页为 <http://www.openstack.org/community>，可以在上面检索到很多公开的资源 and 信息。

1.5.1 邮件列表

对于任何一个开源社区来说，邮件列表都是最为重要的一环。

OpenStack 邮件列表由多个子邮件列表组成，分别用于讨论不同的话题，起到不同的作用。如果读者对某类话题感兴趣，不论是潜水，还是要加入论战，都应该订阅相应的邮件列表。

其中，openstack@lists.openstack.org 是 OpenStack 社区通用的邮件列表。一般来说，寻求帮助或是发布信息都可以通过这个邮件列表。

openstack-announce@lists.openstack.org 是专门用来接收 OpenStack 发布团队和 OpenStack 安全团队重要通告信息的一个邮件列表，比如有关 OpenStack 的发布、OpenStack 解决了最新某个安全漏洞等信息。

openstack-operators@lists.openstack.org 是为 OpenStack 运维人员相互讨论而建立的一个

平台，各公司以及各云计算运营商的运维人员可以在这个邮件列表里交流关于 OpenStack 安装、部署和运维方面的经验。

`openstack-dev@lists.openstack.org` 为 OpenStack 开发者交流技术而设立，是一个对于 OpenStack 开发者而言相当重要的邮件列表。比如，某个开发者打算开发一个新的功能，但开发之前希望能够知道社区对这个功能是否感兴趣和是否认可，或者其他人能否提出更好的建议等，这类问题都可以通过这个邮件列表进行讨论。此外，在开发调试过程中遇到的任何问题，都可以在这个邮件列表中提出。如果有其他人也遇到过类似问题或知道如何解决，通常都会热心地给予回答。

由于 OpenStack 中项目众多，为了防止邮件泛滥，通常在发送邮件时会在标题加上项目名称作为前缀以示区别，比如“[nova]”表示这个邮件的内容是关于 Nova 的，“[nova][neutron]”表示既与 Nova 有关，也与 Neutron 有关，如此一来，开发者可以更有针对性地阅读邮件。

此外，`openstack-qa@lists.openstack.org` 用于 QA 讨论测试案例设计、测试配置以及测试项目；`openstack-security@lists.openstack.org` 用于讨论 OpenStack 安全问题；`foundation@lists.openstack.org` 由 OpenStack 基金会专用；`user-committee@lists.openstack.org` 供 OpenStack 用户委员会讨论问题使用。所有的邮件列表都可以在 <https://wiki.openstack.org/wiki/MailingLists> 找到。

1.5.2 IRC 和项目例会

邮件列表无法做到与社区即时互动，我们必须等待有人看到邮件并作出答复，所以它更适合于抛出一个问题或想法，系统地阐述一个观点，并不需要立即得到其他人的帮助和建议。

对于与社区在线即时交互，OpenStack 社区推荐使用 IRC。所有的 OpenStack IRC 聊天室（Channel，又称为频道）可以在 <https://wiki.openstack.org/wiki/IRC> 上找到。比如，`#openstack-dev` 用于讨论开发过程中遇到的问题，`#openstack-nova` 则用于讨论有关 Nova 的相关问题。

我们提交 patch 之后，其他开发者会对其进行代码评审并给出意见，随后我们进行回复。然后经过一段时间，可能是若干天，之前提出意见的开发者会对阅读到我们的回复以及修改后的 patch，这样的周期可能会比较长，此时，我们就可以在 IRC 上进行即时讨论，提高 patch 评审效率。

有些时候，我们会发现，patch 提交后 OpenStack 的集成测试系统在进行集成测试时出错，我们就可以进入 `#openstack-infra` 频道向常驻在那里的集成测试维护者寻求帮助并报告问题。

还有一类比较特别的频道，比如 `#openstack-meeting`，专门为各个项目举办例会而设。由于各项目例会的时间基本上都不一样，所以共用这一个频道。实在有冲突时再开设其他频道，比如可以使用 `#openstack-meeting-alt` 或者 `#openstack-meeting-3`。

OpenStack 各项目例会的安排都列在 <https://wiki.openstack.org/wiki/Meetings>。对于 OpenStack 开发者来说，参加项目例会是重要的。例会一般由该项目的 PTL 主持，鼓励该项目的开发者参加并讨论会议议题。例会是 OpenStack 开发者融入社区的一个相当好的

机会。

IRC 频道和项目例会固然很好，但它们也有局限性。比如，由于开发者比较忙，并不是都会经常上线，实时地解决别人遇到的问题。再比如，由于 OpenStack 是一个全球性的项目，很多比较活跃的开发者的上线的时候，往往是在中国的深夜，这种情况下，邮件列表能起到一个互补的作用。此外，由于时差的原因，想要参加一些项目例会并不是一件容易的事，在这种情况下，我们只有靠查阅会议记录和日志来弥补。

1.5.3 Summit 和 Meetup

OpenStack 每半年发布一个版本，发布期间会举办 Summit 进行庆祝，并同时围绕下一个版本中的新设计和新功能展开讨论。

可以说每一届 OpenStack Summit 都是 OpenStack 社区的一次盛会，来自全球各地的开发者、测试人员、运维人员、学术研究人员和其他爱好者与会者聚集一堂，庆祝 OpenStack 的发布，交流各自使用 OpenStack 过程中的经验，学习和研究 OpenStack 这一发布版中的新技术，讨论并设计下一个版本中的新功能，寻求 OpenStack 发展的新机遇。

在 OpenStack Summit 举办之前，社区会在邮件列表里以提名和投票的方式决定下一个 OpenStack 发布版的代号。代号的规则是按照从 A 到 Z 为首字母的顺序，并通常以举办 OpenStack Summit 所在地的某个城市或地区命名。

表 1-1 所示为到目前为止，每一次 OpenStack Summit 的举办地点以及相应 OpenStack 发布版的代号。

表 1-1 OpenStack 发布版代号和 Summit 举办地

OpenStack 发布版代号	OpenStack Summit 举办时间	OpenStack 发布版发布时间	OpenStack Summit 举办地	发布版代号的意义
Austin	2010 年 7 月	2010 年 10 月	Austin, Texas, 美国	Texas 州首府 Austin 市
Bexar	2010 年 11 月	2011 年 2 月	San Antonio, Texas, 美国	举办地的 Bexar 郡
Cactus	不详	2011 年 4 月	不详	Texas 州的 Cactus 市
Diablo	2011 年 4 月	2011 年 9 月	Santa Clara, California, 美国	举办地 California 州湾区附近 Diablo 市
Essex	2011 年 10 月	2012 年 4 月	Boston, Massachusetts, 美国	举办地附近 Essex 市
Folsom	2012 年 4 月	2012 年 9 月	San Francisco, California, 美国	举办地附近 Folsom 市
Grizzly	2012 年 10 月	2013 年 4 月	San Diego, California, 美国	举办地 California 州州旗中的一个元素
Havana	2013 年 4 月	2013 年 10 月	Portland, Oregon, 美国	举办地 Oregon 州的一个非自治社区名称
Icehouse	2013 年 11 月	2014 年 4 月	香港, 中国	举办地香港的“雪厂街”的英文名称

续表

OpenStack 发布版本号	OpenStack Summit 举办时间	OpenStack 发布版发布时间	OpenStack Summit 举办地	发布版代号的意义
Juno	2014 年 5 月	2014 年 10 月	Atlanta, Georgia, 美国	举办地 Georgia 州的一个地名
Kilo	2014 年 11 月	2015 年 4 月	Paris, 法国	国际单位中质量基本单位千克 Kilogram 的发源地是法国巴黎
Liberty	2015 年 5 月	2015 年 10 月	Vancouver, 加拿大	加拿大萨斯喀彻温省的一个乡村名
Mitaka	2015 年 11 月	2016 年 4 月	东京, 日本	日本东京城市圈三鹰市市名
Newton	2016 年 4 月	2016 年 10 月	Austin, 美国	奥斯汀东 9 街国家史迹名录上登记的房屋
Ocata	2016 年 10 月	—	Barcelona, 西班牙	巴塞罗那北部乘列车 20 分钟可达的一个海滩名

由于 OpenStack 起创公司之一的 Rackspace 位于美国得克萨斯州 Texas，所以最初的 OpenStack 代号基本上都以德州地名命名，而且 Summit 的举办时间也不是很有规律。此后，随着更多公司和开发者的加入，Summit 的举办形成了每半年一次的模式，分为春季（一般在 4~5 月份）和秋季（一般在 10~11 月份）两个时间段。

2013 年 11 月，OpenStack 基金会首次将 Summit 放在美国之外的地区举办，因为看重中国和亚洲地区的市场前景，第一站选择的是中国香港，而且当时，中国、印度、日本及东南亚的 OpenStack 用户和开发者规模已经不落后于北美和欧洲地区。此后，2014 年 11 月 Kilo Summit 的举办地点选择了法国巴黎，2015 年 11 月 Mitaka Summit 的举办地点选择了日本东京，2016 年 10 月 Ocata Summit 的举办地点选择了西班牙的巴塞罗纳，2017 年秋季将选择澳大利亚的悉尼，OpenStack Summit 的全球化将是未来的趋势。

关于 Summit 上的演讲，OpenStack 基金会和 Summit 的主办方也有严格的挑选流程。一般情况下，每次 Summit 基金会都会收到 1000 多个演讲申请，而由于时间的限制，不得不挑选出 25%~35% 的名额，大部分的演讲者和演讲申请都会被遗憾地拒之门外。

至于谁可以被选中，谁会落选，主题主席（Track Chair）们有着绝对的权利。一般来说，Summit 的演讲分成很多主题，比如关于存储的、网络的、云计算应用的、社区建设的等。每一个主题都会有 3~4 名主题主席负责审阅演讲申请，并一起决定批准或拒绝申请。在演讲申请提交截止之后，社区会发动一轮投票和拉票，即广泛收取社区会员意见，看哪些演讲申请会足够吸引观众。值得一提的是，投票票数不是主题主席选择该演讲的唯一标准，而只是参考指标，主题主席完全可以根据自己的判断标准来决定最终演讲名单。

我们知道主题主席对于演讲有很大的权利，但是 OpenStack 基金会招募主题主席也是相当严格的，基金会和组委会会根据社区个人自愿申请，公司、地区、社区角色等多样性，以

及在该主题领域专业程度等多因素综合考量主题主席的人选，以保证公平性和多样性。来自国内去哪儿公司的 OpenStack 大使叶璐和 EasyStack 公司的郭长波都曾经分别担任过 Community Building 主题和 Upstream Development 主题的主题主席。

除了 Summit 之外，各地区各项目也会不定期地举办技术交流研讨会，我们称之为 Meetup。比如，在春秋两季 Summit 之间，Nova 社区一般会举办 Nova 中期 Meetup，与会者以开发者为主，人数在几十人至一百人以内，均围绕 OpenStack 的功能设计展开讨论。

在中国的北京、上海以及西安，OpenStack 中国社区也会联合一些公司不定期地举办技术交流会，讨论的议题有许多类型，包括 OpenStack 部署、开发以及未来构想等。

1.5.4 其他社交平台

除了以上所介绍的沟通渠道，OpenStack 还有其他一些社交平台，比如：

- 推特 Twitter: @openstack;
- 脸书 Facebook: <http://www.facebook.com/pages/OpenStack/104139126308032?v=wall>;
- LinkedIn: <http://www.linkedin.com/company/openstack>。

https://wiki.openstack.org/wiki/OpenStack_User_Groups 上列出了全球各地 OpenStack 用户兴趣小组的联系方式和联络人。比如，在国内，有一个称为“中国 OpenStack 用户组 (China OpenStack User Group)”的小组，经常会不定期举办各种讨论交流的研讨会，他们有自己的微博账号@COSUG，也会在他们的网页上发布 Meetup 信息。这些 Meetup 有些是介绍新发布的 OpenStack 具有哪些新特性，有些是介绍企业部署最佳实践和运维经验分享，有些是介绍 OpenStack Summit 见闻，有些则是庆祝 OpenStack 生日等。

1.6 其他开源项目

OpenStack 作为一个开源项目自诞生起就不缺少竞争对手的陪伴，这其中影响力最大的有 CloudStack、Eucalyptus 和 OpenNebula。表 1-2 所示为到目前为止 OpenStack 与其他开源项目之间的一个简单的比较。

表 1-2 OpenStack 与其他开源项目比较

	OpenStack	CloudStack	Eucalyptus	OpenNebula
第一版发布时间	2010 年 10 月	2010 年 5 月	2008 年 5 月	2008 年 3 月
版本	Apache	Apache	GPLv3	Apache
开发语言	Python	Java、C	Java、C	C++、C、Ruby、Java、Shell 脚本、lex、yacc 等
宿主机是否支持 Linux	是	是	是	是
宿主机是否支持 Windows	是	否	否	

续表

	OpenStack	CloudStack	Eucalyptus	OpenNebula
宿主机是否支持裸机	是	是	是	否
客户机是否支持 Linux	是	是	是	是
客户机是否支持 Windows	是	是	是	是
是否支持 Xen	是	是	是	是
是否支持 KVM	是	是	是	是
是否支持 VMware	是	是	是	是
主要支持厂商	Rackspace、Red Hat、IBM、HP、Mirantis、VMware、Intel、NEC、Huawei、Cisco、Canonical、SUSE、Dell 等	Citrix、Intel、Red Hat、IBM、SUSE、Cisco 等	Amazon、Dell、HP、Intel、Mallanox、Novell、Red Hat、VMware 等	IBM、CERN、Logica 等
代表用户	NASA、Canonical、RackSpace、Intel、AT&T、Paypal、新浪等	塔塔集团、阿朗、英国电信、GoDaddy、韩国电信、中国移动、中国电信、国家电网等	索尼、Puma、趋势科技、NASA、InfoSys、中国工商银行等	德国电信、RIM、SARA、中科院、中国移动研究院等

(1) CloudStack

在 2008 年前后, 美籍华人梁胜联合其他几个创始人创立了一家公司, 开发出一款名为 VMOps 的云平台系统, 这便是 CloudStack 的前身。后来, 这家公司收购了互联网域名 Cloud.com, 将公司更名为 Cloud.com, 同时将 VMOps 改名为 CloudStack, 希望将来致力于云计算解决方案的研发。

2010 年 5 月, Cloud.com 公司将大部分 CloudStack 的代码在 GPLv3 版权下发布成免费软件, 只保留大约 5% 不进行公开。

2011 年 7 月, Citrix 收购了 Cloud.com 公司希望丰富自己的云产品线。同年 8 月, Citrix 在 GPLv3 版权下发布了剩下的 CloudStack 代码。

2012 年 2 月, Citrix 发布了 CloudStack 3.0。同年 4 月将 CloudStack 版权修改成 Apache 2, 并将其完全捐献给 Apache 软件基金会, 放入到 Apache 孵化器中。

2013 年 3 月, CloudStack 在发布 CloudStack 4.0.2 之后, 终于从 Apache 孵化器中成功毕业成为 Apache 软件基金会的顶级项目 (Top-Level Project, TLP)。

到目前为止, CloudStack 有着众多的商业客户, 包括 GoDaddy、英国电信、日本电报电话公司、韩国电信、中国电信和 Autodesk 等。

(2) Eucalyptus

Eucalyptus (Elastic Utility Computing Architecture for Linking Your Programs to Useful Systems)最初是美国加利福尼亚大学圣芭芭拉分校计算机科学学院的一个研究项目。2009年, Benchmark Capital 公司出资 550 万美元创立了 Eucalyptus System 公司,并将研究项目商业化。不过, Eucalyptus 仍然按照开源项目进行维护和开发,只是 Eucalyptus System 会在开源的基础上构建额外附加的功能,并提供客户支持服务。

所以, Eucalyptus 本质上是开源的,只是同时还有收费的企业版。Eucalyptus 经历了近 5 年的发展,目前最新为 4.0.1 版本。2014 年 9 月,惠普公司宣布收购 Eucalyptus 成为其旗下产品,加强其云战略。

(3) OpenNebula

OpenNebula 起源于 2005 年,当时它还是一个由两位研究员 Ignacio M. Llorente 和 Ruben S. Montero 主导的开源研究项目,研究的目的是致力于找到一个 IaaS 的云计算解决方案。该方案能够提供开放、灵活、可扩展的管理中间层,用户通过该管理层能轻易地自动生成和编配虚拟数据中心。

2008 年 3 月, OpenNebula 发布了第一个公开版本。在 OpenNebula 后续的发展过程中,社区用户起到了很大的作用。OpenNebula 中的很多功能,都是社区用户在分布式架构上部署大规模虚拟机之后,为了解决在使用过程中的不便之处以及满足新的业务需求而实现的,可以说, OpenNebula 的很多功能是开源社区智慧和创新的结果。

2010 年 3 月, OpenNebula 的主要贡献者创立了 C12G 实验室并为企业用户提供附加值服务,且承诺为 OpenNebula 提供持久的维护工作。OpenNebula 商业化运作自此开始。同时, C12G 实验室还管理着开源网站社区 OpenNebula.org。

2013 年 9 月, OpenNebula 迎来了它的第一次社区大会,当时来自世界各地的顶尖公司和组织都参加了这次盛会。2014 年 8 月, OpenNebula 发布了新的稳定版本 4.8,这是目前最新稳定版。

除了上述一些 OpenStack 的竞争对手,在整个云计算生态圈里,还有其他一些项目正在成为或者已经成为业界瞩目的热点,比如在 SDN 网络领域和分布式存储领域中有影响力的几个项目,以及基于容器技术的一些云平台管理技术等。

(1) OpenDaylight

在网络技术的版图里, SDN 技术早已是业界公认的未来方向,但是怎样统合各个厂商甚至学术界的标准,一直以来是个难题。在 2013 年,一大批传统的 IT 设备厂商联合几家软件公司,发起了 OpenDaylight 项目,简称 ODL。此项目发展至今,已经成为开源的 SDN 方案中最有影响力的项目之一。从 2013 至今, ODL 已有 4 个正式的发行版本,而每一个版本都是以化学元素表中某一元素来命名,充满了学术气息。这 4 个版本的名字依次是:氢、氦、锂、铍 (Beryllium)。

在 ODL 社区的成员公司中,分为白金、黄金、白银不同等级,区分了赞助费用和权益的不同。其中白金会员有博科、思科、Intel、Citrix、戴尔、爱立信、惠普、IBM、微软、红帽。

在社区的运作中，也充分强调了 ODL 技术决策中的开放性和公正性，保证了项目发展的活力和健康。

在技术上，ODL 项目有其独到之处。首先，此项目是基于 Java 开发平台的，充分利用了 Java 平台上成熟的动态模块技术（OSGI），以微服务架构为基础，非常灵活和高效地集成了各种插件来提供多家厂商设备的支持，以及各种高级网络服务。在北向 API 接口的设计上，ODL 不但提供了 Restful 的 API，也提供了函数调用方式的 OSGI 接口，以应对不同方式的北向集成方案。在南向 API 接口的设计上，充分考虑了多协议支持和多厂商设备的适配的便利。而与 OpenStack 中 Neutron 模块的集成，一直是 ODL 项目的技术重点之一，最近也有突飞猛进的发展。

（2）OPNFV

顾名思义，OPNFV 项目的初衷是提供一个 Open 开放开源的电信运营级 NFV 方案，其中 OP 指代的是 Open Platform，而 NFV 是 Network Function Virtualization 的缩写。此项目 2014 年创立至今，已有 4 个发行版本，最新的是 2016 年 4 月份的 2.0（Brahmaputra）。

OPNFV 项目更像一个其他 NFV 架构相关项目集成、测试、优化的标准，整合了包括 OpenStack 在内的 30 多个项目，比如 OpenDaylight、KVM、Xen、Ceph 和 OVS 等。OpenStack 是 OPNFV 方案中最关键的项目。作为云计算基础资源的管理者，可谓是 NFV 栈上的中心环节。

OPNFV 项目对于已有技术没有自己的代码库，但强调“上游优先”的原则，所以 OPNFV 社区已经为上游软件做了大量的贡献，比如说 OpenStack 的众多项目。同时 OPNFV 社区也积极同其他标准组织如 ETSI 合作，共同推进 NFV 领域的业界标准。

（3）Ceph

在云计算的整体架构里，SDS 存储方案的制定，也是无法绕过的要点。现如今，Ceph 项目应该是开源分布式存储方案的最主流选择，尤其是和 OpenStack 配合部署的场景下更是如此。

Ceph 诞生于学术项目，在原作者依此创业的公司被红帽公司于 2014 年收购之后，红帽公司成为 Ceph 项目的最主要贡献者。此后在 2015 年，Ceph 正式开始社区化管理，社区委员会吸收了 8 家成员公司，包括红帽、Intel、Suse、Cisco、Canonical、Fujitsu、CERN 和 SanDisk。

Ceph 项目提供了全方位的分布式存储接口，涵盖了对象存储、块存储和文件系统全部 3 种云环境使用场景。在 OpenStack 的云环境中，由于分布式块存储方案往往集成 Ceph 的 RDB 技术，所以在选用对象存储方案时，很多用户更倾向于同样适用 Ceph 的对象存储服务。因此，Ceph 对于 OpenStack 原生的 Swift 的推广有所影响。

对于云计算客户生产环境的要求，纯社区版的 Ceph 还在性能上有一定的差距，所有国内外出现了很多基于 Ceph 社区版的商业定制化产品，从性能优化、稳定性、用户界面等方面都有大幅度的提升。

（4）Docker

2013 年 Docker 项目横空出世，并在此后的发展过程中对计算机业界产生了深远的影响。当然，在 Docker 诞生之前，各种容器相关的基础技术已经成熟，比如 Linux 内核中 cgroup 和 namespaces 已经广为人知，而联合文件系统技术也已有多种选择，在各种 UNIX/Linux 系统中，类容器项目也早已被广泛使用，比如最原始的 UNIX 中的 chroot 技术，到 Solaris 中的 Zones 技术，还有 FreeBSD 中的 jail，Linux 系统中则是 OpenVZ 和 LXC 等项目。但为什么 Docker 能够取得前所未有的巨大成功？天时地利人和的原因之外，纯粹从技术层面来看，Docker 真正抓住了用户的痛点，进而设计出了直达使用者期望的友好使用界面，尤其是对于软件开发更是如此，所以 Docker 的流行首先是发生在 DevOps 领域，其后逐渐波及其他领域。这里的友好界面不单单指提供给用户的命令行指令，也包括了对于容器映像的管理策略，和统一的在线仓库。

Docker 映像制作和发布的便利，也彻底改变了很多对于软件发行方式的看法。不管是否把容器技术用于云计算环境的搭建，人们使用 Docker 作为自己软件产品的持续集成交付（CICD）方案的基础，或使用 Docker 仓库作为发布渠道之一，已日渐流行。所以 OpenStack 的软件发行管理，部署方式，以及各个组件的开发调试，都可以从中受益。

相比于虚拟化技术，以 Docker 为代表的容器技术在做到应用隔离的同时，却没有带来性能的损失，这是一个很明显的技术优势。所以在注重计算性能的云计算场景，部分用户更倾向于使用容器作为底层基础技术。但是，安全性却是容器技术的短板，毕竟运行于同一主机的不同容器实例间共享一个内核，安全漏洞防不胜防。当然，很多安全隔离增进技术也层出不穷，从不同的角度来弥补可预期的安全隐患。

Docker 作为一个创新性的项目，也激发了其他厂商的灵感，基于多种原因，多家厂商也在力推自己的相似技术，比如 CoreOS 的 rkt 等。而 Docker 公司自己也因为商业的原因，有意严控 Docker 项目的设计和走向，时而会造成一些业界的分歧和困惑。所以在其他厂商强烈的期盼下，OCI（Open Container Initiative）项目和社区应声而立，以开发中立的容器技术标准。最终这些项目和标准的前景如何，还有待观察。

（5）Mesos

Mesos 是 2009 年在 UC Berkeley 创立的项目，其目的是作为一个集群环境中资源隔离、共享、调度的统一平台技术，就如 Mesos 自己所强调的，它可以被看作是分布式系统层面的操作系统。在 2015 年，Mesosphere 公司成立，其商业目标是基于 Mesos 技术，提供产品级质量的数据中心分布式操作系统，即 DC/OS。

Mesos 统管的资源包括计算（CPU）、内存、IO 和文件系统，并能够做到细粒度又十分高效的资源隔离与调度。作为一个分布式系统的管理平台，Mesos 在容错性上也表现出色。而其双层调度设计和开放的计算框架接口，使得 Mesos 可以方便地适配不同的分布式计算应用，典型的应用场景包括 Hadoop 和 Spark 为代表的大数据处理平台，或大规模的数据挖掘应用，或方兴未艾的深度学习应用框架，等等。

在 Docker 出现之前，Mesos 以发展和流行多年，而 Docker 所带来的功能，正是 Mesos

所亟需的，所以这两个项目相辅相成，从而放大了 Mesos 商业应用的前景，催生了一批国内外基于此技术的创业公司。

(6) Kubernetes

Kubernetes 简称 k8s，Google 所创建的开源项目，是 Google 内部使用的 Borg 系统的开源版本，现在从属于 CNCF（Cloud Native Computing Foundation）社区，以保证其公立性。

从一开始 Kubernetes 就把自己定位为容器云环境的管理软件，从整体系统的设计上充分考虑了容器（主要指的是 Docker）技术的特点，这也是它和其主要竞争对手 Mesos 的不同点之一。

Kubernetes 在资源的定义上，以容器为基础元素，又引入了“Pod”的概念，把运行相同应用的多个容器实例看作一个管理和调度单元（Pod）。而一个 Pod 需要运行在单个 Minion 之中，Minion 可以理解成一个主机（Host）。在集群中有多个 Minion，被中央控制节点 Master 统一管理。而在 Pod 的基础上，Kubernetes 又抽象出 Service 的概念，是多个 Pod 一起工作提供服务的抽象。

总而言之，Kubernetes 引入非常多的新抽象概念，而其集群管理中所要解决的调度、高可用、滚动升级等问题，都是围绕此来进行设计。

Kubernetes 是容器云计算环境的基石之一，被业界寄予厚望，因此而生的初创公司在国内外并不鲜见。

1.7 OpenStack 的技术发展趋势

OpenStack 自诞生起，经历了同期类似项目的竞争，受到了来自新兴技术的挑战，有时其看似缓慢低效的社区开发模式也被人诟病，但还是取得了如今云计算中流砥柱的地位。而且在可预期的未来，随着云计算技术需求的进一步拓展，相信 OpenStack 还会继续扮演越来越重要的角色。

OpenStack 在技术上也从未停下创新脚步，在不停地吸收新的元素，从而更加灵活高效和健壮。

(1) 裸机资源的管理

OpenStack 在架构设计上最初是以虚拟机的管理为中心的，但随着云计算用户使用场景多元化的趋势，对于虚拟机之外的计算资源的管理与编排就越发显得必要，比如裸机和容器。

对于裸机计算资源的管理，目前来看还相当的不成熟，虽然 Nova 社区近来十分重视统一化计算资源模型的努力，但是刚刚起步，现在唯一可以做到了的是一种藉由 IroniC 的协助所实现的一种比较僵化的资源调度，远远达不到实际用户对于裸机资源的灵活管理。而对于裸机资源特定的功能需求，目前尚处于社区讨论阶段，比如说 RAID 的配置、物理网口资源的细化管理、网卡绑定配置，而日后是由 Nova 实现还是放在 IroniC 中处理，甚至由新起项目来完成此任务，尚待观察。唯一确定的是，此类用户需求总有一天会出现在 OpenStack 中。

此外，OpenStack 中的资源调度器（现在是 Nova 的调度器，但有计划独立成一个新的项目），也会逐渐做到根据裸机硬件配置的差异进行更加精准的调度和编排。进一步来看，随着 OpenStack 对于裸机资源管理的日渐成熟，Nova 和 Ironic 等项目的进一步发展，纯粹的“裸机云”方案也会大行其道。

（2）容器化

Docker 的横空出世，为众多开发者带来了非常多的便利，此后出现的 Kubernetes 之类的容器集群的管理平台，更是让容器技术在云计算的领域大放异彩，OpenStack 的发展也从中受益匪浅，比如在 OpenStack 控制层服务的容器化方面：开发环境的容器化、打包和发布的容器化、部署和升级的容器化、云环境动态伸缩的容器化，等等。

在云计算概念已广为人知的今天，一个新的应用从设计之初就要考虑是否“云原生（Cloud Native）”的问题。而不幸的是，OpenStack 中众多服务却不是云原生的架构，而随着容器化的努力，很多难题都一一迎刃而解，比如单节点中不同服务运行环境隔离的问题、版本升级和回滚的难题、OpenStack 控制面服务的动态伸缩部署和高可用处理。

此外，容器技术降低了对于特定操作系统的依赖，同时容器化的部署方案还可以提供给使用者一个简单而优雅的界面，大大提高了用户体验。这些努力都集中在 OpenStack 大帐篷里的 Kolla 项目（包括 Kolla 的几个子项目，如 Kolla-kubernetes 等），以及 Stackanetes 等独立项目。目前来看，除了技术层面的分析，业界各个 OpenStack 厂商的产品路线也都印证了此趋势。相信在 OpenStack 的部署和控制面服务的管理等方面，容器化是不可逆转的方向。

（3）多元化的云计算环境

与 OpenStack 所主导的以虚拟化技术为中心的云计算技术路线相比，以容器为计算资源基础单元的容器云是一股新兴的力量，以 Kubernetes、Mesos 和 Docker 公司的 Swarm 等项目为代表。

这两种不同的云计算路线未来的趋势应该是相互渗透相互借鉴的合作方式，如前所述的 OpenStack 部署容器化则是明证之一。再者，OpenStack 作为 IT 基础架构资源（Infrastructure）的成熟管理方案，对于网络和存储方向的技术积累，可以帮助容器云快速成熟，而容器云的项目也不必再重造轮子。

而从云计算环境的最终用户需求来看，根据计算模型的不同，用户希望灵活地去申请和调度不同类型的计算资源，包括虚拟机、裸机、容器，这样就势必要把 OpenStack 和容器云融合在一起才是一个完备的解决方案。

这里我们可以设想这样一种模式：既然 OpenStack 可以通过 Kubernetes 等容器云技术来部署和管理控制面服务，那 OpenStack 则可以被看作是 Kubernetes 容器云中的一个云原生的应用，随之而来的就是如云原生应用般的灵活管理，而直接面对裸机资源的 Kubernetes 又可以通过集成 Ceph、Swift、Cinder、Neutron 等项目来完善对网络和存储资源的池化管理。

（4）更加彻底的资源池化

云计算概念中被大家所熟知的一个就是资源的池化，包括技术资源的池化、网络资源的

池化（SDN）和存储资源的池化（SDS），总括来说就是所谓的 SDI（Software Defined Infrastructure，软件定义基础架构）。

众所周知，主流的服务器硬件配置基本上是固定的，但以 Intel 主导的 RSD（Rack Scale Design）整机柜服务器技术，则可以做到动态调配整个机柜中的 CPU、内存、存储、网口等硬件资源，灵活地组成不同配置的服务器。此项技术把资源池化的概念实现得更加彻底，所带来的好处也不言而喻。

OpenStack 开发基础

如何才能成为一名 OpenStack 开发者，与众多大牛共舞？本章的内容希望能够给你一些提示和帮助。

2.1 相关开发资源

我们开发水平的高低取决于我们手中所掌握的资源。人类文明近万年，计算机历史不到百年，云计算火了近 10 年，即使 OpenStack 也已经存在了 5 年，相关知识早已浩瀚无边，我们所能接触到的不过大海之一滴，所能理解掌握谨记于心的更不过是这一滴中很微小的一部分，我们每日所做的也不过是为了使这一滴更大，使那一部分更多而已。

如果把这一滴比作我们所掌握的资源，那么我们平时开发时应该不求下笔千行，而力求迅速地从各种资源中找到解决的方案。举个例子，比如开发一个视频监控系统，那么我们可以从 sourceforge 上的项目 MPEG4IP 得到很多灵感。

下面列举一些 OpenStack 开发中常用的资源以供读者参考。

2.1.1 OpenStack 社区

如第 1 章所述，OpenStack 有一个紧密团结了众多使用者和开发者的 OpenStack 社区，那里是大牛们的战场，小牛们的天堂，任何一个 OpenStack 开发者都可以从中受益匪浅。

不同于 Linux 内核社区中 LKML (Linux Kernel Mailing List) 对问题讨论、代码 review 等的大包大揽，邮件列表的核心地位在 OpenStack 社区中被一定程度地弱化，代码的 review 工作将通过 Gerrit 系统来完成，同时，答疑功能也由一个专门的问答网站 “Ask OpenStack” (<https://ask.openstack.org/en/questions/>) 来分担。

这个问答网站的风格类似于 Stack Overflow (<http://stackoverflow.com/>)，能够对认为正确的解答投票，而且有中文和英文两个站点点可以选择。和所有的开源社区一样，我们在询问问题时都应该保持一个良好的习惯，即先在邮件列表的存档中搜索一下，查看自己想提的问题是否已经被讨论并回答过了。

2.1.2 OpenStack 文档

评估软件质量的重要标准之一是其可维护性，而是否有完善的文档、文档是否同步则是

影响可维护性的一个重要因素。

软件的开发者无外乎两类，第一类人维护自己的代码，但是随着时间的流逝，以前的理解和记忆已经随风而散了，每次修改 bug 或增加新功能时，需要通过查看文档和代码来帮助开发者回忆当时的开发场景。

第二类人，是那些代码不是自己编写的，而需要阅读相关文档的代码，甚至为了弄明白某一处代码的意图，需要了解整个模块的逻辑流程，还要查看相关的需求以及设计文档，但是很遗憾，这些文档一般都和实际代码不同步。

无论我们属于哪一类人，文档的重要性都是不言而喻的。每一个项目，每一个模块，甚至每一行代码都有自己的故事。这些故事应该被留存在自己的历史档案里，而大部分的时间我们只是它们的过客而已。

但软件开发的现实是：几乎没有任何一个软件，在其生命周期中，均由最初的开发人员维护；几乎没有任何一个软件，在其生命周期中，它的文档与代码是保持同步的。

对于开源项目来说，文档的缺乏与滞后更为突出，但即使是最为挑剔的批判者，也不得不承认 OpenStack 在这方面有着较为突出的改进，它甚至创建专门的项目来维护 OpenStack 相关的各类文档。

为了有针对性地适应不同参与者的需求，OpenStack 也将自己的文档进行了相应的分类，比如针对开发者的、针对维护人员的、针对各行各业的使用者的。虽说表面看起来比较分散，甚至有点凌乱，但自有其内在规律。而这个规律的源头就是一个 wiki (<https://wiki.openstack.org/wiki/Documentation>)，以及 2012 年上半年 summit 上的一个有关文档的 session (<https://etherpad.openstack.org/p/folsomdocsplanning>)。

这个 wiki 中指出 OpenStack 的文档主要位于 3 处。

- wiki.OpenStack.org：主要包含项目文档、会议记录等信息，在这里用户可以清楚地了解如何向 OpenStack 做贡献、OpenStack 代码是如何进行管理等。
- docs.OpenStack.org/developer/<projectname>：针对开发者的文档，由位于各个项目源码目录中 doc/source/子目录下的 rst 文件生成。
- docs.OpenStack.org：主要包含针对维护人员、使用者等其他参与者的官方文档，比如安装指南（Installation Guides）、终端用户指南（End User Guide）等，内容由 OpenStack Manuals 项目生成。

2.1.3 OpenStack 书籍

本质上，学习 OpenStack 开发就是学习 OpenStack 的源代码，任何 OpenStack 有关的书籍都是基于源码，而又不高于源码的。

因此，在阅读本节内容之初，我们必须端正自己的认识：源码才是中心，书仅仅，也只能充当促进或辅助我们理解源码的作用。

待到山花烂漫时，还是那些经典在微笑。Linux 成长逾二十年，才给我们留下了经典

LKD/ULK/LDD/..., 而 OpenStack 满打满算也不过 4 岁有余, 书是涌现了那么一些, 但是否经典尚只能说“不可说”。

在这些有限的 OpenStack 书籍里, 可以归纳为两类: 一类是针对部署与操作的, 比如 O'REILLY 的《OpenStack Operations Guide》和《Deploying OpenStack》; 另一类是侧重框架与原理的, 即针对开发者的。目前已有的书籍里除本书之外很难说有哪一本可以完全归到这一类, 大多也只是有部分内容涉及而已。

按照本书的预期以及主张的 OpenStack 开发学习过程, 针对 OpenStack 开发的书籍也应该分为两类: 第一类是介绍 OpenStack 及各组成部分框架与实现原理的书籍, 帮助读者了解 OpenStack 的设计实现特点, 对其有个整体全局的认识和理解。第二类是专注于 Open 某个组成部分或项目, 如果希望对 OpenStack 不是泛泛而谈而是有更深入的理解, 则可以选择一个自己感兴趣的项目, 仔细分析它的代码, 不懂的地方就通过社区、邮件列表等途径弄懂, 切勿得过且过, 这样分析下来, 对 OpenStack 很多通用的机制也同样会非常了解。“一通则百通”就是这个道理。

2.1.4 其他网络资源

除了前面所提到的, 还有很多其他的论坛或网站值得我们关注。下面列举其中的一部分供参考。

- <https://wiki.openstack.org/wiki/BloggersTips>: 该 wiki 页面收集了许多有价值的 blog 链接。
- <http://status.openstack.org/release/>: 该页面可以看到全部项目的进展。
- <http://planet.openstack.org/>: OpenStack 开发者的 blog 聚合。
- <http://www.mirantis.com/blog/>: OpenStack 系统集成商 Mirantis 的 blog。Mirantis 是 OpenStack 的最大赢家之一, 于 2015 年 8 月 24 日获得了英特尔领投的 1 亿美元融资, 并且 OpenStack 的最大用户, 几乎都是他的客户, 包括思科 Webex、ebay 等。

Mirantis 把自己的部分培训教程开源, 比如“OpenStack 2 天培训”等, 网上可以找到相应的资料。

- <http://www.sebastien-han.fr/blog/>: eNovance 公司成员 Sebastien Han 的 blog。
- <http://programmerthoughts.com/>: Swift PTL (Project Technical Lead) John Dickinson 的 blog, 文章不多, 不过可以了解很多 Swift 的基本知识。
- <http://www.cloudbase.it/blog/>: Cloudbase 实现了 OpenStack 对 Hyper-V 的支持。

2.2 OpenStack 开发的技术基础

学习 OpenStack 是一项浩大的工程, 需要具备以下一些基础知识。

(1) Python 编程

Python 是 OpenStack 的主要开发语言，因而也自然而然地成为每一个 OpenStack 开发者必备的语言基础。

当然，如果我们之前使用的开发语言并不是 Python，比如 C，我们也并不需要首先对 Python 掌握到非常精深的程度才去接触 Python 的代码，本质上它与 C、Java、Perl、Ruby 等还是属于同一类型的语言。我们可以在浏览 OpenStack 源码的过程中学习 Python 语言以及各种 Python 的用法，且用且学，毕竟我们绝大多数人都应该不是从无到有地去构建一个 OpenStack 的项目。

(2) Linux 环境编程

目前为止，OpenStack 仍只被部署在 Linux 系统上，它的开发环境自然也基于 Linux，那么能够熟练使用 Linux 操作系统在 Linux 环境下进行编程开发便成为一个基本要求。

此外，掌握一些操作系统中比较基础的理论，也会对我们的理解带来额外的益处，比如进程的概念、CPU 和内存的关系等。

(3) 网络基础

参与一个云计算平台的开源项目，有一定的网络基础知识是不言而喻的事情。如果是 Neutron，则对网络知识储备有更高的要求。

(4) 虚拟化

虚拟化技术是云计算的基石，比较好的理解虚拟化技术对我们理解 OpenStack 的很多逻辑关系非常有帮助。

(5) 版本管理工具 (Git)

Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。Linus 把 Git 介绍为 “The stupid content tracker”。越来越多的著名项目采用 Git 来管理项目开发，包括 OpenStack、Android、Rails 等。网上也能够找到很多的使用教程，甚至还有专门针对 Git 的培训。

除了上述各项基础要求，对数据库、软件架构设计等的了解也非常有必要。

2.3 部署开发环境

对开源项目来说，所谓的开发环境应该包括两个部分：一是源码，二是运行测试源码的环境。

如果只是希望了解实现原理与框架，那么从源码仓库下载源码，通过合适的浏览工具阅读即可。如前所述 OpenStack 使用 Git 管理源码，这就意味着我们起码要能够使用 Git 工具。

而如果希望能够成为一个开发者并贡献自己的代码，则搭建一个运行测试自己代码的环境是必须的。对于 OpenStack，这个部分的工作通常由 devstack 来完成。

2.3.1 Git

要成为一名 OpenStack 开发者，贡献自己的代码，就必须能够与其他众多的开发者协同工作，因此，熟练使用 Git 管理代码便成为一个必备的基础条件。

1. Git 的由来

Linux 于 2002 年 2 月开始使用 BitKeeper 作为 Linux 内核的版本控制工具。但是 BitMover 公司在商业版的 BitKeeper 之外，提供的 BitKeeper 只是仅可免费使用但不允许加以修改开放的精简版，因此，包括 GNU 之父 Richard Stallman 之内的很多人，对 Linux 使用 BitKeeper 感到不满。

然而，当时市场上并没有其他具备 BitKeeper 类似功能的自由软件可用，于是有些人就尝试对其进行逆向工程，这惹恼了 BitMover，于是该公司决定停止提供 BitKeeper 的免费版本。为解决无工具可用的窘境，Linux 便自行开发 Git，希望在适当的工具出现前，暂时充当解决方案。当时 Linux 曾称 Git 为愚蠢的内容管理器（the stupid content tracker）。当 Git 有了迅速成长之后，Linux 就建议能够以其作为长期的解决方案，并于之后的 2.6.12-rc3 内核第一次采用 Git 进行发布。

2. 一段录像

在 Git 历史上有一段很著名的录像，是 Linux 在 Google 的一个演讲，我们可以在 Youtube 上看到它。在这段录像中，Linux 说明了设计 Git 的原因、基本的设计哲学，以及与其他版本控制工具的比较。

从技术的观点上，Linux 非常尖锐地批判了 CVS 与 SVN。虽然 Linux 从来没有使用过 CVS 去管理内核代码，但是他在商业公司曾有过一段不短时间的使用经历，而且对其强烈地厌恶。同时他批判 SVN 是毫无意义的，因为 SVN 尝试从各方面去改善 CVS 的一些缺点，却无法根本地解决一些基本的使用限制。具体来说，就是 SVN 改善了创建分支所耗费的成本，相对 CVS 利用了比较少的系统资源，却无法解决合并分支的需求。但是许多项目的开发过程中，都时常需要为不同的新功能创建分支、合并分支，如此一来，SVN 就成为一个没有未来的项目。

Git 作为一个分布式的版本控制工具，可以随意创建新分支，进行修改、测试、提交，这些在本地的提交完全不会影响到其他人，可以等到工作完成后再提交给公共的仓库。这样就可以支持离线工作，本地提交可以稍后提交到服务器上。

3. 一些资源

本书并不会也不需要 Git 的具体使用过程进行介绍，下面仅推荐几个好的网站或资料以供学习参考。

- <http://www.ibm.com/developerworks/cn/linux/l-git/>: 这篇文章详细描述了如何使用 Git

来管理内核代码。

- <http://www.youtube.com/watch?v=4XpnKHJAok8>: 这就是上面提到的那段著名的录像, 相信认真地观看并消化之后会有不小的收获。
- <http://book.opensourceproject.org.cn/versioncontrol/git/gittutorcn.htm>: Git 的中文教程。
- <http://zh-cn.whyyitisbetterthanx.com/#github>: 这是 Kanru 翻译的《为什么 Git 比 X 更好》, 简要地说明了 Git 与其他版本控制工具的比较, 可以让读者了解各种工具之间的差异细节。
- <http://github.com/>: 很多人说正是 GitHub 让他们选择了 Git, 相比其他的项目托管网站, 它更像一个社交网络, 可以追踪别人的状态, 不过追踪的不是朋友发出的信息, 而是朋友写出的代码。人们可以在 GitHub 上找到与他们在做的事相关的其他开发人员或项目, 然后轻松地 fork 和贡献, 这样形成了一个以 Git 和各种项目为中心的活跃社区。

4. 获取 OpenStack 源码

OpenStack 源码仓库的镜像有 <http://git.openstack.org/> 和 Github, 两个镜像站点上的代码相同。我们既可以在浏览器中打开 <http://git.openstack.org/cgi> 或 <https://github.com/openstack> 直接浏览, 也可以使用 Git 命令将源码仓库复制一份到本地系统上, 比如:

```
$ git clone git://git.openstack.org/openstack/swift
```

我们也可以从 Launchpad (<https://launchpad.net/<project>>) 上获取稳定的发布版, 比如 Icehouse Nova 对应的链接为 <https://launchpad.net/nova/icehouse/2014.1.1>。

2.3.2 Devstack

Devstack 是一套用来给开发人员快速部署 OpenStack 开发环境的脚本, 它并不适用于生产环境。

使用 Devstack 部署 OpenStack 开发环境, 我们不必再使用 Git 命令手工获取 OpenStack 源码, 因为这是 Devstack 工作的一部分。除此之外, Devstack 自动化部署还包括自动执行所有服务的安装脚本、自动生成正确的配置文件、自动安装依赖的软件包, 如图 2-1 所示。

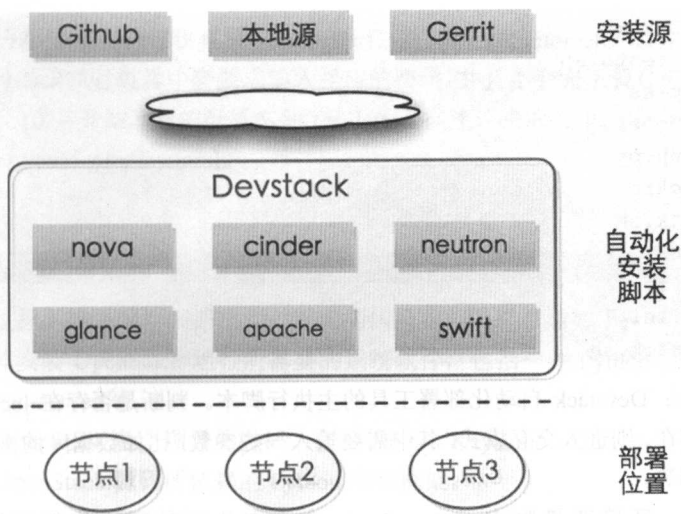


图 2-1 Devstack 自动化部署

1. 获取 devstack 源码

```
$ git clone git://github.com/openstack-dev/devstack.git
$ cd devstack
$ tree -L 1
```

```
.
├── clean.sh
├── data
├── doc
├── exerciserc
├── exercises
├── exercise.sh
├── extras.d
├── files
├── functions
├── functions-common
├── FUTURE.rst
├── gate
├── HACKING.rst
├── inc
├── lib
├── LICENSE
├── MAINTAINERS.rst
├── Makefile
├── openrc
└── pkg
```

```

├── README.md
├── run_tests.sh
├── samples
├── setup.cfg
├── setup.py
├── stackrc
├── stack.sh
├── tests
├── tools
├── tox.ini
└── unstack.sh

```

- **stack.sh**: Devstack 自动化部署工具的主执行脚本。判断是否存在 `localrc` 配置文件，若不存在，则进入交互模式，其中需要输入一些参数，比如数据库的密码、Rabbit MQ 的密码等。
- **openrc**: 环境变量脚本。OpenStack 命令在执行时需要依赖环境变量，比如“`OS_USERNAME`”等，我们可以执行下面的命令导入：

```
$ source openrc admin
```

- **unstack.sh**: 卸载所有已经启动的服务。
- **lib/**: 这个目录下存放了每个服务的自动化安装脚本，比如 `nova`、`swift` 等，包含手动安装时执行的所有命令。

曾经有一个脚本文件 `rejoin-stack.sh` 用于重启所有服务，但是新的版本里它已经被删除 (<http://stackoverflow.com/questions/36268822/no-rejoin-stack-sh-script-in-my-setup>)，如果你仍然需要，可以自己手工创建。

2. 配置 localrc

在 `devstack` 目录下创建一个名为 `localrc` 的文件，代码如下：

```

$ cd devstack
$ vim localrc
# Passwords
MYSQL_PASSWORD=stack
ADMIN_PASSWORD=stack
SERVICE_PASSWORD=stack
RABBIT_PASSWORD=stack
SERVICE_TOKEN=stack

# enable_service ceilometer
enable_plugin ceilometer git://git.openstack.org/openstack/ceilometer
enable_plugin aodh git://git.openstack.org/openstack/aodh

```

这个简单的 localrc 示例仅仅设置了一些密码，并开启了 Ceilometer 服务。如果没有这个 localrc 文件，devstack 执行过程中会要求输入相应的密码，并且不会去下载 Ceilometer 的代码。

从 2013 年 10 月开始，新的配置文件 local.conf 被建议来取代 localrc，更多信息可参阅 <http://devstack.org/configuration.html>。

3. 执行

```
$ ./stack.sh
```

整个执行过程无需干预，根据过程中输出的信息我们可以总结如下：

- 下载并安装 OpenStack 运行所需要的系统软件，包括一些 Python 的组件、MySQL、rabbitmq-server 等。
- 获取 OpenStack 各个项目的源码，包括 Nova、Keystone、Glance、Horizon 等。
- 安装 OpenStack 源码所依赖的 Python 库和框架。
- 安装 OpenStack 各组件。
- 启动各项服务。

整个安装过程所花费的时间依赖于网络状况，中间遇到较多的问题就是某些软件无法下载，但是脚本会比较清楚地报出错误信息，可以将安装出错的软件手动安装，之后重新执行脚本。

stack.sh 脚本成功执行后，Web 服务会被自动启动，我们在浏览器中输入 Devstack 安装所在的服务器地址，就可以打开 Dashboard，如图 2-2 所示。



图 2-2 OpenStack Dashboard 登录页面

首次执行 `./stack.sh` 获得成功之后，可以执行 `./unstack.sh` 关闭所有服务。

`./stack.sh` 脚本实际上会启动一个 `screen session`，实现多个 OpenStack 服务进程共享一个物理终端的窗口管理器。`screen session` 里包括多个 `screen` 窗口，每个都对应一个 OpenStack 服务，比如：

```
5$ n-api* 6$ n-cpu 7$ n-cond 8$ n-crt 9$ n-net 10$ n-sch
```

我们可以运行下面的命令打开 `screen` 窗口：

```
$ screen -ls
There is a screen on:
      8284.stack (2014年07月14日 10时54分19秒) (Attached)
1 Socket in /var/run/screen/S-rqw.
$ screen -x stack
```

从上述 Devstack 的安装过程可知，除了可能的 `localrc` 文件配置，全程并不需要 we 进行干预。下面仅仅是一些可能有益的补充：

- 如果希望在 Xen Hypervisor 的环境下部署 OpenStack，则需要一些额外的设置以及注意事项，具体可参看相关的 wiki。

https://wiki.openstack.org/wiki/XenServer/VirtualBox#Installing_XCP

<https://wiki.openstack.org/wiki/XenServer/DevStack>

- 我们也可以使用一些虚拟机软件，比如 `virtualbox` 等，来安装 Linux 系统并部署 OpenStack 开发环境。

2.4 浏览 OpenStack 源代码

阅读本节的内容之前，我们有必要首先端正一下自己的认识：学习 OpenStack 并不等同于学习 OpenStack 的文档或书籍。OpenStack 的文档确实比较完善，也有一些不错的书籍，但整日地抱着它们用功啃，最多只是说明你是一个很有上进心很应该得到表彰的好青年、好同志，不过也就仅此而已了。

毫不夸张地说，学习 OpenStack 开发就是学习它的代码，代码本身就是最好的参考资料，其他任何经典或非经典的书都只是起到一个辅助作用，不能也不应该取代代码在我们学习过程中的主导地位。

但是面对 OpenStack 这么一个庞然大物，恐惧会在我们心里滋生蔓延。人类进化这么多年，面对复杂的物体和事情总会有天生的惧怕感，体现在 OpenStack 开发学习上就是：那么庞大复杂的代码，让人面对起来，情何以堪啊！

有了这种恐惧无力感存在，心理上就会去排斥面对接触源码，宁愿去抱着各种文档，搜集各种各样五花八门的书籍放在那里囤着，看了又忘，忘了又看，也不大情愿去认真细致地浏览源码。这个时候，我们应该意识到自己需要去克服心理上的脆弱。

有了正确的认识之后，相信本章接下来的内容会使你在浏览学习 OpenStack 源码时，不会迷失在无尽的代码海洋里。

2.4.1 浏览代码的工具

工欲善其事，必先利其器。一个功能强大，同时又使用方便的代码浏览工具对于我们阅读 OpenStack 代码是很有帮助的。下面列举一些常用的代码浏览工具。

1. Vim+各种插件

Source Insight 并没有对应 Linux 的版本。因此对于很多 Linux 初学者来说，在一个完全的 Linux 环境下进行学习，首要解决的一个问题就是，寻找到一个可以取代 Source Insight 的代码浏览工具。

这个工具就是 Vim，各种 Linux 发行版都会默认进行安装。虽然 Vim 默认的编辑界面很普通，甚至说丑陋，但是可以通过配置文件.vimrc 添加不同的界面效果。同时还可以配合 TagList、WinManager 等很多好用的插件或工具，将 Vim 打造成一个不次于 Source Insight 的代码浏览编辑工具。

有关如何将 Vim 打造成一个强大的 Python IDE，可以参考一篇博文：
<http://sontek.net/blog/detail/turning-vim-into-a-modern-python-ide>。

2. Eclipse+PyDev 插件

Eclipse 加上 PyDev 插件应该是 Python IDE 的经典搭配，不过它需要一个 Java 的运行环境，对机器的配置还是稍微有点要求的。如果你用来开发 Python 的机器内存只有 256MB，还是另谋其他出路的好，比如接下来提到的 Spyder。

3. Spyder

Spyder 相对来说比较小众，但也能称得上是一个强大的交互式 Python 语言开发环境，同样提供高级的代码编辑、交互测试、调试等特性，通常各个 Linux 发行版都会内置有它的安装包。

重要的一点是，它还是一个轻量级的选手，比 Eric 等工具轻量得多，启动速度很快，性价比比较高。使用界面如图 2-3 所示。

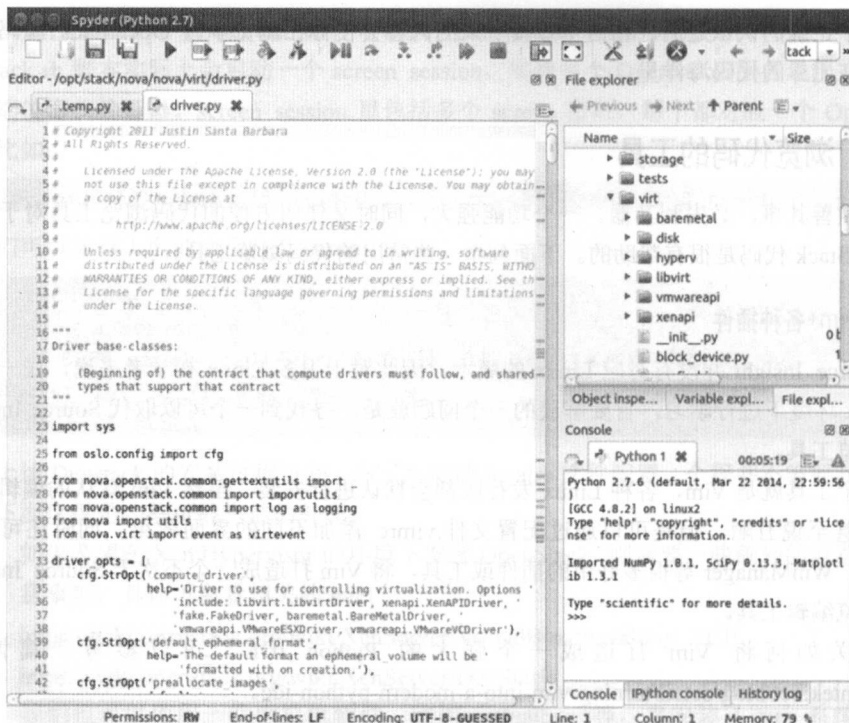


图 2-3 Spyder

2.4.2 分析源码如何入手

既然要学习 OpenStack 源码，就要经常对代码进行分析，而 OpenStack 代码千千万，还前仆后继地不断往里加，这就让大部分人都有种雾里看花花不见的无助感。不过不要怕，孔老夫子早就留给了我们应对之策：敏于事而慎于言，就有道而正焉，可谓好学也已。这就是说，做事要踏实才是好学生、好同志，要遵循严谨的态度去理解每一段代码的实现，多问、多想、多记。如果抱着走马观花、得过且过的态度，结果极有可能就是一边看一边丢，没有多大的收获。

1. 源码地图

我们新到一个城市，一般都是先得到当地的地图以及各种指南。有了它们在手，我们才不至于无头苍蝇般迷惘行走在陌生的街道上。

而我们新接触到一个稍微有点规模的项目，面对铺天盖地袭来的千万代码，内心也总是会渴望有这样的一份地图或指南，让我们能够快捷地对它有个整体的了解并寻找到自己的目的地。

对大多数项目来说，它的编译系统就承担了地图这个很重要的角色，比如 Linux 内核源

码遍布各级目录的 Kconfig 和 Makefile 文件，Android 源码中无处不在的 Android.mk 文件。这些编译系统相关的文件，可以让我们很容易地弄清楚代码之间的脉络，从庞大的代码里定位出我们所关心的部分。

以 Linux 内核简单为例，如果某一天你希望在自己的 Linux 系统里浏览 U 盘里的文件，却发现一直顺从的 Linux 怎么也不能识别出保存有大量珍贵视音频资料的 U 盘，是 U 盘出问题了吗？于是不甘心决定去研究一下内核里 U 盘驱动的实现找找原因。

U 盘是一个 USB 存储设备，你很容易地想到驱动的源码一定是在 drivers/usb/storage/ 目录下，但是当你进入这个目录，看到铺天盖地的数十个文件却不由倒吸一口凉气，于是你安慰自己：这些文件、代码一定不是都与 U 盘的实现相关。

也许，生活中总是充满了跌宕起伏，认真看了一下 Kconfig 和 Makefile 两个文件之后，你的心情明显好了许多。从 Kconfig 文件看，只有 CONFIG_USB_STORAGE 选项与 U 匹配，其他的众多选项明显不是针对 U 盘，只是在混淆我们的注意力，考验我们的心理。接下来再用 CONFIG_USB_STORAGE 选项去 Makefile 文件进行过滤，相关的只剩下了寥寥几个文件，也就是说，我们只需要研究这仅有的几个文件中的部分代码就可以完全理解 Linux 对 U 盘是如何支持的，任务顿时轻松了许多。

但是作为一个解释性语言，遗憾的是 Python 源码并没有这样的一个编译系统来承担 Linux 内核中 Kconfig 与 Makefile 地图的角色。而面对同样纷繁复杂的文件与代码，我们需要寻找新的合适的地图。

2. setup.cfg

如果你使用 devstack 部署整个开发环境，默认情况下，OpenStack 代码安装在 /opt/stack/ 目录下，Nova、Cinder 等各个项目的代码则对应了该目录下的子目录。

走马观花地浏览一下 /opt/stack/ 目录中下载的代码，我们很容易发现每个子项目的源码根目录下都有一个 setup.py 文件和 setup.cfg 文件，凭直觉，我们基本上能够感觉出它们应该就是我们寻找的地图。

如果你已经有一定的 Python 开发基础，你会知道它们是与 Python 模块的分发相关的；如果没有基础的话，你可能需要了解一下 Distutils、Distutils2、Setuptools、Distribute 等 Python 代码分发的工具，这并不是本书的重点。我们首先以 Ceilometer 子项目为例看看 setup.py 的内容。

```
import setuptools

try:
    import multiprocessing # noqa
except ImportError:
    pass

setuptools.setup(
```



```
setup_requires=['pbr'],
pbr=True)
```

这个文件的代码很简单, 仅仅只是调用了 `setup` 函数, 而 `setup` 函数也只是寥寥勾勒两笔。

但事情原本不是这样的, `setup` 函数有大量的参数需要设置, 包括项目的名称、作者、版本等。是 `setup.cfg` 文件的出现将 `setup` 函数解脱出来, 它使用 `pbr` 工具去读取和过滤 `setup.cfg` 中的数据, 并将解析后的结果作为自己的参数。于是, 我们这个故事的重心不可避免地落在了 `setup.cfg` 文件。

`setup.cfg` 文件的内容由很多个 `section` 组成, 比如 `global`、`metadata`、`file` 等, 提供了这个软件包的名称、作者等有用的信息, 但能够帮助及指引我们去更好地理解代码的 `section` 唯有 `entry_points`。

3. entry points

从名称上看, `entry points` 表示入口点, 根据经验, 代码的入口点通常也就是我们理解的突破口, 只要依着这个突破口分析下去, 我们总会理解一个模块或功能的实现原理与细节。

但是 OpenStack 各个子项目中 `setup.cfg` 文件里注册的 `entry points` 非常多, 为了更好地理解, 我们需要明白这些入口点在 Python 项目中的意义。

对于一个 Python 包来说, `entry points` 可以简单地理解为它通过 `setuptools` 注册的外部可以直接调用的接口。仍然以 `Ceilometer` 为例:

```
ceilometer.compute.virt =
    libvirt = ceilometer.compute.virt.libvirt.inspector:LibvirtInspector
    hyperv = ceilometer.compute.virt.hyperv.inspector:HyperVInspector
    vsphere = ceilometer.compute.virt.vmware.inspector:VsphereInspector
    xenapi = ceilometer.compute.virt.xenapi.inspector:XenapiInspector
```

上述代码表示注册 4 个 `entry points`, 它们属于 `ceilometer.compute.virt` 组或者说命名空间 (`entry points group`)。它们表示 `Ceilometer` 目前共实现了 4 种 `Inspector` 从 `Hypervisor` 中获取内存、磁盘等相关的统计信息。

`Ceilometer` 安装后, 其他程序可以利用下面几种方式调用这些 `entry points`。

- 使用 `pkg_resources`:

```
import pkg_resources
def run_entry_point(data):
    group = 'ceilometer.compute.virt'
    for entrypoint in pkg_resources.iter_entry_points(group=group):
        # Grab the function that is the actual plugin.
        plugin = entrypoint.load()
        plugin(data)
```

- 仍然使用 `pkg_resources`:

```
from pkg_resources import load_entry_point
load_entry_point('ceilometer', 'ceilometer.compute.virt', 'libvirt')()
```

- 使用 stevedore，本质上 stevedore 也只是对 pkg_resources 的封装：

```
from stevedore import driver

def get_hypervisor_inspector():
    try:
        namespace = 'ceilometer.compute.virt'
        mgr = driver.DriverManager(namespace,
                                   cfg.CONF.hypervisor_inspector,
                                   invoke_on_load=True)
        return mgr.driver
    except ImportError as e:
        LOG.error(_("Unable to load the hypervisor inspector: %s") % (e))
    return Inspector()
```

这段代码表示，Ceilometer 会根据配置选项“hypervisor_inspector”的设置，加载相应的 Inspector，比如加载 ceilometer/compute/virt/libvirt/目录下的代码去获取虚拟机的运行统计数据。

从上面的代码可以看出，entry points 都是在运行时动态导入的，有点类似一些可扩展的插件，_import_或 importlib 也可以实现同样的功能，但是 stevedore 使这个过程更容易，更有助于我们在运行时动态地导入一些扩展的代码或插件来扩展自己的应用。这种方式也正是 OpenStack 各个子项目所主要使用的。

目前为止，基于对 entry points 的理解，我们可以相对容易地找到所需要研究代码的突破口，比如我们希望研究 Ceilometer 是如何获取虚拟机的内存磁盘等统计数据的，我们就可以根据 ceilometer.compute.virt 这个 entry points 组的定义研究 ceilometer/compute/virt/目录下的代码，甚至可以仿照它下面 libvirt 的实现增加新的 Inspector，对新的 Hypervisor 类型进行支持。

4. console_scripts

在可能的众多 entry points 组中，有一个相对比较特殊的，就是 console_scripts。下面是 Ceilometer 的 setup.cfg 文件对于它的定义：

```
console_scripts =
    ceilometer-api = ceilometer.cmd.api:main
    ceilometer-polling = ceilometer.cmd.polling:main
    ceilometer-agent-notification =
ceilometer.cmd.agent_notification:main
    ceilometer-send-sample = ceilometer.cmd.sample:send_sample
    ceilometer-dbsync = ceilometer.cmd.storage:dbsync
    ceilometer-upgrade = ceilometer.cmd.storage:upgrade
    ceilometer-db-legacy-clean = ceilometer.cmd.storage:db_clean_legacy
    ceilometer-expirer = ceilometer.cmd.storage:expirer
```

```
ceilometer-rootwrap = oslo_rootwrap.cmd:main
ceilometer-collector = ceilometer.cmd:collector:main
```

这里的每一个 entry points 都表示有一个可执行脚本会被生成并被安装，我们可以在控制台上直接执行它，比如 ceilometer-api，因此将这些 entry points 理解为整个 Ceilometer 子项目所提供各个服务的入口点更为准确。

2.5 OpenStack 代码质量保证体系

OpenStack 虽然新生不久，但红火的程度已经吸引了越来越多的眼球，对于这样一个迅速聚集了众多使用者和开发者的开源项目，我们相信，与在另一个行业里同样迅速蹿红的《爸爸去哪儿》不同，“卖萌”一定不是它取胜的关键，它一定有自己一套成熟高效的体系来保证代码的质量与项目的稳步推进。

因此，继如何入手分析源码之后，本节将会对 OpenStack 如何保证自己的代码质量做一些探究。

比尔·盖茨说：“用代码行数来衡量编程的进度，就如同用航空器零件的重量来衡量航空飞机的制造进度一样。”所以，相对于代码的数量，我们通常更乐意去关注代码本身的质量，也因此，在开源社区里，除了某些特殊的目的，我们也更愿意去关注一个人被接受 patch 的数目，而不是这些 patch 里代码的行数。

对于“代码质量”的定义，我们应该每个人都能说个不尽相同的三言两语，但是更多的感觉可能是只可意会不可言传，很难真正地使用统一的标准去定义清楚。当然，已经有一些研究和工具在通过各种指标来对代码的质量进行量化，比如图 2-4 所示。

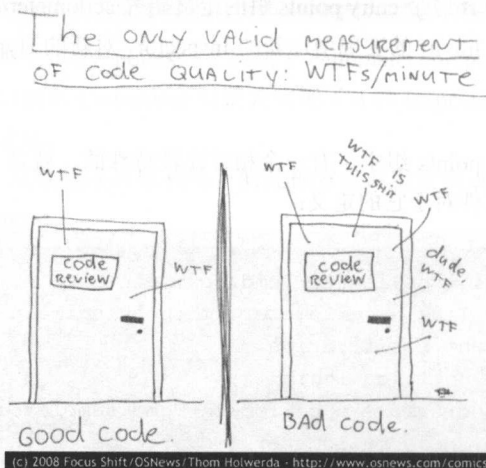


图 2-4 以 Code Review 过程中每分钟出现“脏话”的个数来衡量代码的质量

总之，将“代码质量”定义清楚是一件足够复杂的事情。幸好，笛卡儿很有预见性地在17世纪的某一天，闲极无聊写了这么一本书，书名就叫《方法论》。在这本目前来说绝大部分人都不知道的书里将方法上升到了理论的高度。笛卡儿在他的这本书里将研究问题的方法归纳为简单的一句话，就是“复杂问题要简单化”。

遵循这个方法论，我们这里尝试去解释一下代码质量。

代码一是给计算机读，二是给人读，给维护这份代码的人读。给计算机读比较简单，只要遵守语言的规则，计算机就能将它编译成最后的结果。给人读就比较麻烦，我们去读别人编写的代码的时候，都是希望这个代码写得比较简单，函数很短，命名能够让人望文生义，读起来就像读小说、故事会一样，我们希望的就是我们自己编码的时候应该要做到的目标，这就是站在通俗角度简单化了的代码质量。

这个简单化了的定义强调更多的是代码的可读性，“代码应该是写给其他人来读的，而能让机器运行的仅仅是附带着的。”大牛们如是说。可读性是一切代码质量指标，包括可维护性、可靠性、可扩展性、性能等的基石，一般来说，干净整洁的代码，往往运行起来更快。而且即使它们运转速度不快，也可以很容易地让它们变快。正如人们所说的，优化正确的代码比改正优化过的代码容易多了。

但是对于一个蓬勃发展、前景无限开期的开源项目来说，它的代码质量却不能只是这么简单地给个通俗定义，而是必须有一套行之有效的体系与工具来保证。

站在软件工程的高度，通常来说，代码质量保证的步骤如图2-5所示。

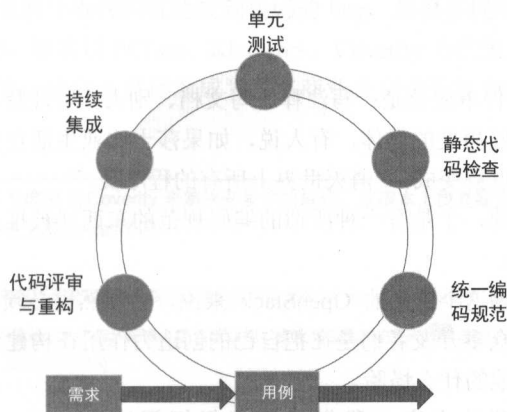


图 2-5 代码质量保证步骤

- 统一编码规范：可读性与可维护性的前提就是一个统一的编码规范。
- 静态代码检查：代码的开发完成以后，接着要进行的工作就是测试。而从计算机理论的角度来说，测试又被划分为静态测试与动态测试。
其中，动态测试指的就是通常意义上我们所说的测试，它会去运行测试代码或直接

运行被测试的软件来发现存在的问题。静态测试则是指应用其他手段实现测试目的，比如属于人工范畴的代码评审（Code Review）与计算机辅助进行的静态代码检查。

- 代码的静态检查主要是指利用静态分析工具对代码进行特性分析，以便检查程序逻辑的各种缺陷和可疑的程序构造，比如不符合编码规范、潜在的死循环等编译器发现不了的错误。之所以称之为静态代码检查，是因为只是分析源代码或者生成的目标文件，并不实际运行源代码生成的文件。它的目的是帮助我们尽可能早地发现代码中存在的问题并及时修复，将其消灭在萌芽状态，就能为后续工作节省大量的花在测试与调试上面的时间。
- 单元测试。
- 持续集成。持续集成（CI，Continuous Integration）是利用一系列的工具、方法和规则，通过自动化的构建（包括编译、发布、自动化测试等）尽快发现问题和错误，来提高开发代码的效率和质量。
- 代码评审与重构。代码评审可以帮助发现静态代码检查过程中无法发现的一些问题，比如代码的编写是否符合编码规范，代码在逻辑上或者功能上是否存在错误，代码在执行效率和性能上是否有需要改进的地方，代码的注释是否完整正确，代码是否存在冗余和重复。通过代码评审发现的问题要通过代码及时解决掉。

本节接下来的内容将会对照上述的代码质量保证步骤，总结 OpenStack 使用的工具与采取的措施。

2.5.1 编码规范

程序员最讨厌的 4 件事应该是：写注释、写文档、别人不写注释、别人不写文档。那么对于这样一个貌似很不好相处的群体，有人说，如果莎士比亚生活在当下，他会是一名科技作家，而且他的座右铭也会变成：“消灭世界上所有的程序员。”

消灭当然是做不到的，于是有一种所谓的编码规范的东西就被推上了前台，来预防程序员的各种个性与创造力。

对于达到百万行代码这个量级的 OpenStack 来说，它当然也必须有自己的一套编码规范来约束以及预防自己的众多开发者们是在把自己的创造力作用在构建一个蓬勃发展的开源云项目上，而不是一个其他的什么怪胎。

至于这个编码规范的内容，我们尽可能仔细阅读 <http://legacy.python.org/dev/peps/pep-0008/>。本节的内容将着重放在 OpenStack 编码规范检查工具，及其相关的一个子项目 Hacking 上。

1. 代码静态检查

如前所述，代码静态检查是代码质量保证体系里很关键的一环，编码规范的检查又是代码静态检查的其中一种。为了更好地理解后面的内容，我们有必要先对代码静态检查做一番

了解。

对于 C、C++ 等编译型语言来说，因为可以与编译器进行比对，理解代码静态检查会更为容易一些。

编译器与代码静态检查工具都能检查代码中的潜在问题。一般地，编译器最重要的作用是生成可执行文件，所以对于词法语法的分析相对局部一些，即在检测错误时，前后查看的代码较少。这也是基于编译器的性能来考虑的。因为还有很多的优化要做，所以编译器在词法语法分析上不能耗费太多时间。尤其是 Android 这种代码量很大的项目，编译都要花费半天甚至更多的时间时，5% 的性能差距就比较多了。

所以说，尽管编译器擅长发现一些错误，但通常会考虑速度而放弃了对一些较难发现的条件的检查。这样一些原本可以发现的错误，经常会被遗漏而成为应用中的 bug，比如未成功释放已分配的内存、死循环等。

但是代码静态检查工具并没有性能以及时间上比较苛刻的需求，因为它们并不用像编译器那样需要频繁运行，所以它们可以牺牲运行的速度去换取更为彻底的词法语法分析，更为准确地找到更多的问题。

使用代码静态检查工具，付出性能与时间的代价，收获就是把更多原本只有测试阶段，甚至应用阶段才能发现的 bug 在编码阶段暴露出来，在 bug 的成本上分析，有这样一个公认的结论：bug 发现得越晚，修正的成本就越高，测试阶段修正 bug 的成本是编码阶段的约 4 倍的关系。

在编码阶段，静态的分析代码就能找到代码的 bug，是很多程序员简单而又美好的梦想。这个梦想在 21 世纪初，随着以 PCLint、Klocwork、Coverity 为代表的开源或商业静态分析软件的出现而变成了现实。这些工具逐渐成为很多商业公司或开发者的居家必备之良药，如图 2-6 所示。

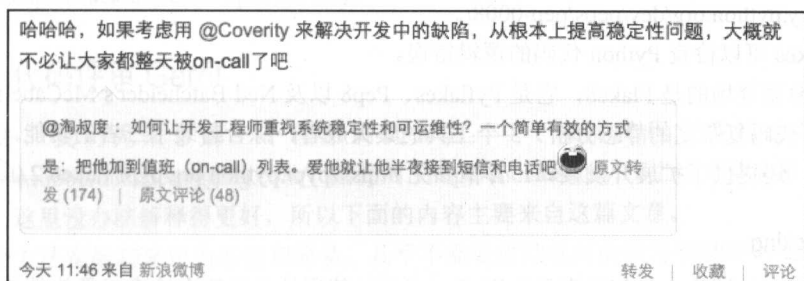


图 2-6 代码静态检查工具

对于开源世界来说，2006 年由美国国土安全部发起并与斯坦福大学合办了一个关注开源代码完整性的研究项目 Coverity Scan，这个项目每年都是会使用 Coverity 公司的代码静态检查工具评估和帮助改进包括 Linux 内核、Android、Python 在内的主要开源项目的代码质量，比如在 2006 年就帮助修复了 6000 多个 bug。同时，这个项目每年都会出一份开源软件质量报

告。图 2-7 所示为 2011 年报告中有关各种缺陷比例的一部分。

Defect Type	Number of Defects	Percentage
NULL Pointer Dereference	6,448	27.95%
Resource Leak	5,852	25.73%
Unintentional Ignored Expressions	2,252	9.76%
Use Before Test (NULL)	1,867	8.09%
Buffer Overrun (statically allocated)	1,417	6.14%
Use After Free	1,491	6.46%
Unsafe use of Returned NULL	1,349	5.85%
Uninitialized Values Read	1,268	5.50%
Unsafe use of Returned Negative	859	3.72%
Type and Allocation Size Mismatch	144	0.62%
Buffer Overrun (dynamically allocated)	72	0.31%
Use Before Test (negative)	49	0.21%

图 2-7 开源软件质量报告

2. Python 代码静态检查 Flake8

对于 OpenStack 息息相关的 Python 代码静态检查来说，目前的工具主要有 Pylint、Pep8、Pyflakes、Flake8 等。

Pylint 违背了 Python 开发者 Happy Coding 的倡导，或许这也是未被 OpenStack 社区所采纳的缘故。

至于 Pep8 则备受 Python 社区所推崇，负责 Python 代码风格的检查。据路边社报道，某些公司招聘 Python 工程师的要求有一条就是代码符合 Pep8 标准，具体可参见 <http://legacy.python.org/dev/peps/pep-0008/>。

Pyflakes 可以检查 Python 代码的逻辑错误。

最后粉墨登场的是 Flake8，它是 Pyflakes、Pep8 以及 Ned Batchelder's McCabe script（关注 Python 代码复杂度的静态分析）3 个工具的集大成者，综合封装了三者的功能，在简化操作的同时，还提供了扩展开发接口，详情参见 <https://pypi.python.org/pypi/flake8/2.0>。

3. Hacking

有了上文的铺垫，我们很容易知道 OpenStack 使用的代码静态检查工具就是 Flake8，并实现了一组扩展的 Flake8 插件来满足 OpenStack 的特殊需要。这组插件被单独作为一个子项目而存在，就是 Hacking。

```
flake8.extension =
    H000 = hacking.core:ProxyChecks
    H101 = hacking.checks.comments:hacking_todo_format
    H102 = hacking.checks.comments:hacking_has_license
```



```

H103 = hacking.checks.comments:hacking_has_correct_license
H104 = hacking.checks.comments:hacking_has_only_comments
H105 = hacking.checks.comments:hacking_no_author_tags
H106 = hacking.checks.vim_check:no_vim_headers
H201 = hacking.checks.except_checks:hacking_except_format
H202 = hacking.checks.except_checks:hacking_except_format_assert
H203 = hacking.checks.except_checks:hacking_assert_is_none
H231 = hacking.checks.python23:hacking_python3x_except_compatible
H232 = hacking.checks.python23:hacking_python3x_octal_literals
H233 = hacking.checks.python23:hacking_python3x_print_function
H234 = hacking.checks.python23:hacking_no_assert_equals
H235 = hacking.checks.python23:hacking_no_assert_underscore
H236 = hacking.checks.python23:hacking_python3x_metaclass
H237 = hacking.checks.python23:hacking_no_removed_module
H238 = hacking.checks.python23:hacking_no_old_style_class
H301 = hacking.checks.imports:hacking_import_rules
H306 = hacking.checks.imports:hacking_import_alphabetical
H401 = hacking.checks.docstrings:hacking_docstring_start_space
H403 = hacking.checks.docstrings:hacking_docstring_multiline_end
H404 = hacking.checks.docstrings:hacking_docstring_multiline_start
H405 = hacking.checks.docstrings:hacking_docstring_summary
H501 = hacking.checks.dictlist:hacking_no_locals
H700 = hacking.checks.localization:hacking_localization_strings
H903 = hacking.checks.other:hacking_no_cr
H904 = hacking.checks.other:hacking_delayed_string_interpolation

```

从上面 Hacking 源码中的 setup.cfg 文件内容可以看出,到目前为止,Hacking 主要在注释、异常、文档、兼容性等编码规范方面实现了将近 30 个 Flake8 插件,详情参见 <http://docs.openstack.org/developer/hacking>。

2.5.2 代码评审 Gerrit

首先,我们先来了解一个程序调试大法——“橡皮鸭程序调试法”,它来自 <http://www.rubberduckdebugging.com/>,酷壳网 (<http://coolshell.cn/>) 上也有篇文章用中文进行了阐释,这里没办法解释得更好,所以下面的内容主要来自这篇文章。

这个方法实施起来相当方便和简易,几乎不需要借助任何的软件和硬件的支持,可以随时随地地实验,你甚至可以把程序打印出来,在纸面上进行调试。

那么,为什么这个方法叫做橡皮鸭呢?一是因为橡皮鸭子貌似是西方人在泡澡时最喜欢玩的一个小玩具,所以,这个东西应该家家户户都必备的。二是这个方法由西方人发明,所以,就被取名为“橡皮鸭”了。下面是整个调试方法的流程,如图 2-8 所示。



图 2-8 橡皮鸭调试法

- 1) 找一个橡皮鸭子。你可以去借、去买或自己制作……反正你要有一个橡皮鸭子。
- 2) 把这个橡皮鸭子放在跟前。标准做法是放在你的桌子上、电脑显示器边，或是键盘边，反正是你的跟前，面朝你。
- 3) 打开你的源代码，不管是电脑里的还是打印出来的。
- 4) 对着那只橡皮鸭子，把你写下的所有代码，一行一行地，精心地，向这只橡皮鸭子解释清楚。记住，这是解释，你需要解释出你的想法、思路 and 观点。不然，那只能算是表述，而不是解释。
- 5) 当你在向这只始终保持沉默的橡皮鸭子解释的过程中，你会发现你的想法、观点、或思路 and 实际的代码相偏离了，于是你也就找到了代码中的 bug。
- 6) 找到了 bug，一定要记得感谢一下那个橡皮鸭子。

这个方法是否让你感觉太“愚蠢”、太“弱智”了？不过，这个方法的确有效。因为，这就是“Code Review”的雏形！它的核心思想可以概括为：一旦一个问题被充分地描述了他的细节，那么解决方法也是显而易见的。

相信各位都有过这样的经历，当你死活都找不到问题的原因的时候，当你寻求他人的帮助时，对别人解释整个你的想法和意图或是问题背景的时候，你自己都没有解释完，就已经找到问题的原因了。这就是这个方法的意义所在。

所以，“橡皮鸭”只是一个形式，其主要目的是要你把自己写的代码做“自查”，也就是自己解释给自己听。当然，为了不让你像个“精神分裂”的程序员，引入“橡皮鸭”是很有必要的（虽然这样还是有点精神病，但比起精神分裂来说算是好的了）。所以，真实的本质是 Code Review。

那么，对于 OpenStack 来说，为了保证代码评审的有效进行，首先需要做的是为我们寻找合适的道具“橡皮鸭”，然后提供一个将这些道具和我们有效联结起来的平台。

这些道具自然就是分散在全球各地的 OpenStack 开发者们，与橡皮鸭不同的是，他们会发表自己的意见和看法。而这个联结的平台就是 Gerrit。

1. Gerrit 工作流程

Android 在 Git 的使用上有两个重要的创新：一个是为多版本库协同而引入的 `repo`（对 `git` 使用的封装），另一个就是 Gerrit——代码审核服务器。Gerrit 为 Git 引入的代码审核是强制性的，也就是说除非特别的授权设置，向 Git 版本库的推送（Push）必须要经过 Gerrit 服务器，修订必须经过代码审核的一套工作流程之后，才可能经批准并纳入正式代码库中。

OpenStack 也将 Gerrit 引入到自己的代码管理里，工作流程大体和 Android 对 Gerrit 的使用相同，区别是过程更为简洁，而且使用了 Jenkins 来完成自动化测试，如图 2-9 所示。

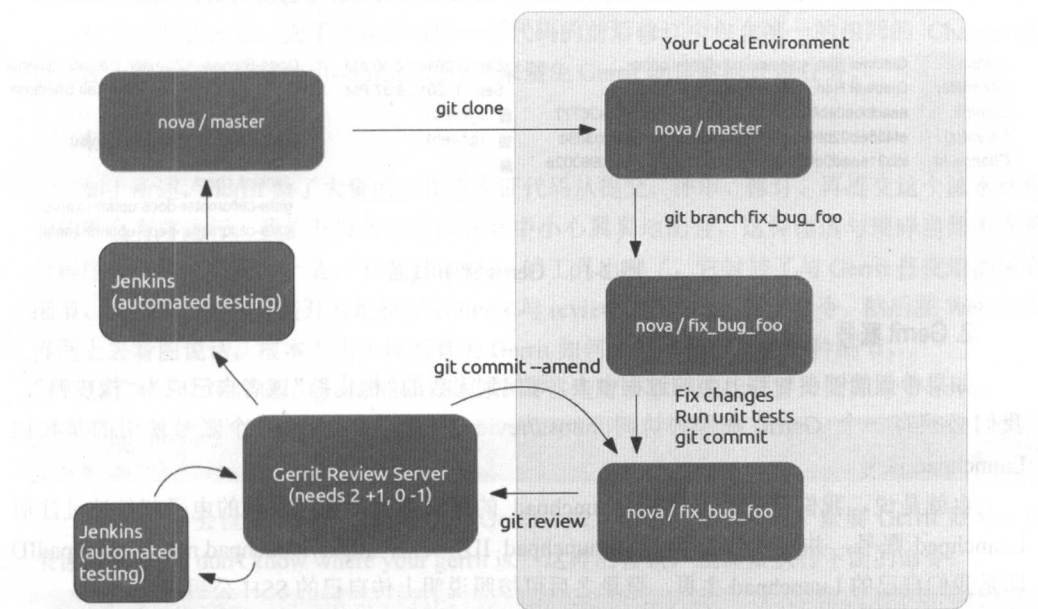


图 2-9 Gerrit 工作流程

首先我们在本地代码仓库中做出自己的修改，然后我们能够很容易通过 `git` 命令（或 `git-review` 封装）将自己的代码 `push` 到 Gerrit 管理下的 Git 版本库。Jenkins 将对我们的提交进行自动化测试并给出反馈，其他开发者也能够使用 Gerrit 对我们的代码给出他们的注释与反馈，其中，项目的 maintainer（OpenStack 中称为 Core Developer）的反馈权重更高（+2），如果你的 patch 能够得到两个“+2”，那么恭喜你，你的 patch 将被 merge 到 OpenStack 的源码树里。

所有这些注释、质疑、反馈、变更等代码评审的工作都通过 Web 界面来完成，因此 Web 服务器是 Gerrit 的重要组件，Gerrit 通过 Web 服务器实现对整个评审工作流的控制。图 2-10 所示为针对某一个提交的评审页面。

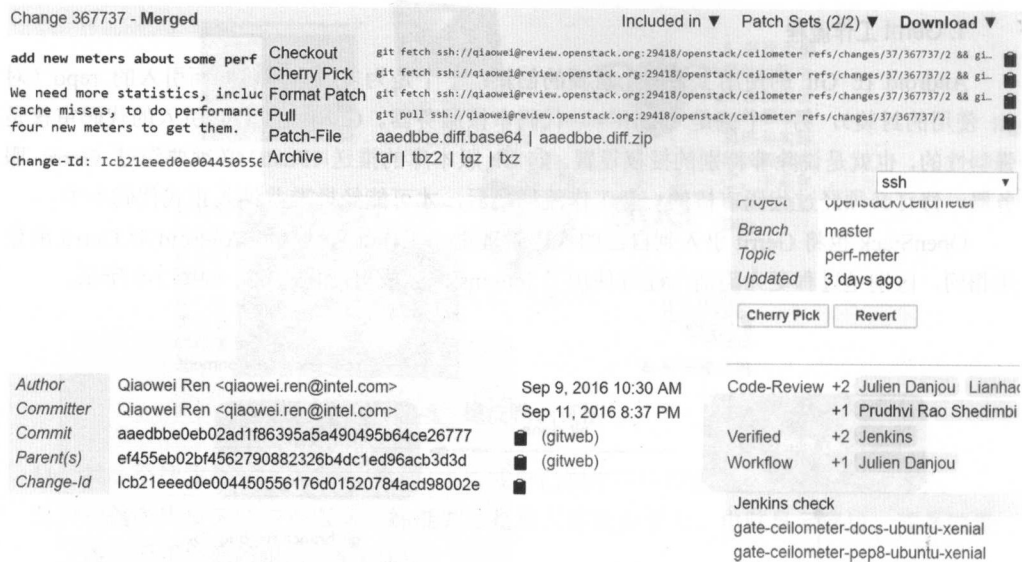


图 2-10 Gerrit 评审页面

2. Gerrit 账号

如果希望能够参与到上面的过程中去，寻找到无数的“橡皮鸭”或者自己成为“橡皮鸭”，我们必须有一个 Gerrit 账号去访问 <https://review.openstack.org/>，这个账号使用的是我们 Launchpad 账号。

也就是说，我们首先需要访问 Launchpad 的登录页面，使用自己的电子邮件地址注册 Launchpad 账号，并为自己选择一个 Launchpad ID，之后 <https://launchpad.net/~LaunchpadID> 即是我们自己的 Launchpad 主页，登录之后可按照说明上传自己的 SSH 公钥。

使用 Launchpad 账号登录之后，我们还需要上传自己的 SSH 公钥（SSH public key），公钥设置的页面有相应的 HowTo 告诉我们如何生成公钥并上传。

3. Gerrit 实现原理

Gerrit 基于 SSH 协议实现了一套自己的 Git 服务器，这样就可以基于自己的需求对 Git 数据传递进行更为精确的控制，为上述工作流程的实现建立了基础。访问 https://review.openstack.org/ssh_info 可以查看到这个 Git 服务器的域名和端口“review.openstack.org 29418”，我们可以发现它使用了端口 29418，并非是标准的 22 端口。

Gerrit 的 Git 服务器，只允许用户向特殊的引用 refs/for/<branch-name> 下执行推送(push)，其中<branch-name>即为开发者的工作分支。Gerrit 会为新的提交分配一个 task-id，并为该 task-id 的访问建立引用 refs/changes/nn/<task-id>/m，比如图 2-10 中所示的 refs/changes/37/367737/2，其中：

- task-id 为 Gerrit 顺序分配给该评审任务的全局唯一的号码。

- nn 为 task-id 的后两位数，位数不足用零补齐，即 nn 为 task-id 除以 100 的余数。
- m 为修订号，该 task-id 的首次提交修订号为 1；如果该修订被拒绝，需要更新代码后重新提交，修订号会依次增加。

为了保证在代码修改后重新提交时，不会产生新的重复的评审任务，Gerrit 要求每个提交包含唯一的 Change-Id，Gerrit 一旦发现新的提交包含了已经处理过的 Change-Id，就不再为该修订创建新的评审任务和 task-id，而是仅仅把它作为已有 task-id 一次修订。

比如图 10-7 所示评审任务的 Change-Id 为 Change-Id: Icb21eed0e004450556176d01520784acd98002e，在它被 merge 到正式的 OpenStack 源码树前共有两次修订“Patch Set 2/2”。

对于开发者来说，为了实现针对同一份代码的前后修订中包含唯一的相同的 Change-Id，需要在执行提交命令时使用--amend 选项，来避免 Gerrit 创建新的评审任务。

4. git-review

如上所述，Gerrit 做了大量的工作来保证代码从提交、评审、修订、再提交这个流水线作业的顺利有序进行，我们开发者也需要在其中小心翼翼地配合，这种谨慎与琐碎当然不太符合程序员们的审美观，于是一个叫 git-review 的工具出现了，它封装了与 Gerrit 打交道的所有细节，我们需要做的只是开心地执行 commit 与 review 这两个 git 的子命令，然后在 Web 图形界面上去看图说话，根本不用去琢磨有关 Gerrit 如何“吃喝拉撒”的种种细节。

为了对一个项目使用 git-review，我们需要首先对该项目进行设置，比如对于 nova：

```
$ cd nova
$ git review -s
```

git-review 会检查用户是否能够登录 Gerrit，如果不能，它会向用户索要 Gerrit 账号。如果你看到“We don't know where your gerrit is.”这样的错误，就需要执行下面的命令：

```
$ git remote add gerrit ssh://<username>@review.openstack.org
:29418/openstack/nova.git
```

然后我们最经常做的事情，除了修改代码之外，就是按照一个熟练工的标准执行下面的这些命令：

```
$ git checkout -b branch-name
$ git commit -a (--amend)
$ git review
```

至于 git-review 如何去安装在此不作赘述，因为大多数 Linux 发行版已经将其集成到了自己的包管理器。

2.5.3 单元测试 Tox

StackOverflow 上一个有 16.7k 分的人问了个有关单元测试的问题“How deep are your unit

tests?”，意思就是说“单元测试需要做多细？”，或者换句话说“单元测试的这个单元的粒度是怎样的？”

针对这个问题，下面的回答获得了压倒性的票数，被评为最佳回答。

I get paid for code that works, not for tests, so my philosophy is to test as little as possible to reach a given level of confidence (I suspect this level of confidence is high compared to industry standards, but that could just be hubris). If I don't typically make a kind of mistake (like setting the wrong variables in a constructor), I don't test for it. I do tend to make sense of test errors, so I'm extra careful when I have logic with complicated conditionals. When coding on a team, I modify my strategy to carefully test code that we, collectively, tend to get wrong.

老板为我的代码付报酬，而不是测试，所以，我对此的价值观是——测试越少越好，少到你对你的代码质量达到了某种自信(我觉得这种的自信标准应该要高于业内的标准,当然,这种自信也可能是种自大)。如果我的编码生涯中不会犯这种典型的错误(如:在构造函数中设了一个错误的值),那我就不会测试它。我倾向于去对那些有意义的错误做测试,所以,我对一些比较复杂的条件逻辑会异常小心。当在一个团队中,我会非常小心地测试那些会让团队容易出错的代码。

翻译来自酷壳网的文章,这不重要,重要的是这个回答来自于 Kent Beck (敏捷开发 XP 与测试驱动开发 TDD 的奠基者)。下面是有人针对 Kent 这个回答的调侃。

The world does not think that Kent Beck would say this! There are legions of developers dutifully pursuing 100% coverage because they think it is what Kent Beck would do! I have told many that you said, in your XP book, that you don't always adhere to Test First religiously. But I'm surprised too.

只是要地球人都不会觉得 Kent Beck 会这么说啊!我们有一大堆程序员在忠实地追求着 100%的代码测试覆盖率,因为这些程序员觉得 Kent Beck 也会这么干!我告诉过很多人,你在你的 XP 的书里说过,你并不总是支持“宗教信仰式的 Test First”,但是今天 Kent 这么说,我还是很惊讶!

回到 OpenStack,单元测试又被称为 Small Tests,粒度并不以开发者个人意愿以及对自身水平的自信而转移,起码从形式上追求着几近 100%的代码覆盖率。也因此,我们提交一些新的代码时,必须要做的事情是往往提交更多的测试代码,而且常常花在单元测试上的时间要更多。

1. OpenStack 单元测试

概括来说,OpenStack 的单元测试追求的是速度、隔离以及可移植。对于速度,需要测试代码不和数据库、文件系统交互,也不能进行网络通信。另外,单元测试的粒度要足够小,

确保一旦测试失败，能够很容易迅速地找到问题的根源。可移植是指测试代码不依赖于特定的硬件资源，能够让任何开发者去运行。

单元测试的代码位于每个项目源码树的<project>/tests/目录，遵循 oslo.test 库提供的基础框架。通常单元测试的代码需要专注在对核心实现逻辑的测试上，如果需要测试的代码引入了其他的依赖，比如依赖于某个特定的环境，我们在编写单元测试代码的过程中，花费时间最多的可能就是如何隔离这些依赖，否则，即使测试失败，也很难定位出问题所在。

对 SUT（the system under test，被测试系统）完成隔离的基本原则是引入 Test Double（类似特技替身演员），用 Test Double 来替代测试中的每一个依赖。

有多种类型的 Test Double，比如 Mock 对象、Fake 对象等，它们可以作为数据库、I/O，以及网络等对象的替身，并将相应的操作隔离，在测试运行过程中，当执行到这些操作时，不会深入方法内部去执行，而是直接返回我们假设的一个值。

```
import mock
from oslotest import base

from ceilometer.compute.virt import inspector as virt_inspector
from ceilometer.compute.virt.xenapi import inspector as xenapi_inspector

class TestXenapiInspection(base.BaseTestCase):

    def setUp(self):
        api_session = mock.Mock()
        xenapi_inspector.get_api_session =
mock.Mock(return_value=api_session)
        self.inspector = xenapi_inspector.XenapiInspector()

        super(TestXenapiInspection, self).setUp()

    def test_inspect_instances(self):
        vms = {
            'ref': {
                'name_label': 'fake_name',
                'other_config': {'nova_uuid': 'fake_uuid', },
            }
        }

        session = self.inspector.session
        with mock.patch.object(session, 'xenapi_request',
                                return_value=vms):
            inspected_instances = list(self.inspector.inspect_instances())
            inspected_instance = inspected_instances[0]
```



```
self.assertEqual('fake_name', inspected_instance.name)
self.assertEqual('fake_uuid', inspected_instance.UUID)
```

上述代码示例使用了 Mock 对象替换 XenServer 环境下 XenAPI 网络连接对象,如此一来,我们就可以在 KVM 等任何环境下执行单元测试,而不局限于 XenServer 环境。

2. Tox

执行单元测试的途径有两种: Tox 或者项目源码树根目录下的 run_tests.sh 脚本。不过通常我们使用的都是 Tox。

Tox 是一个标准的 virtualenv (Virtual Python Environment Builder) 管理器和命令行测试工具。它可以用于检查软件包能否在不同的 Python 版本或解释器下正常安装;在不同的环境中运行运行测试代码;作为持续集成服务器的前端,大大减少测试工作所需时间。

每个项目源码树的根目录下都有一个 tox 配置文件 tox.ini, 比如 Ceilometer 项目的 tox.ini 片段:

```
[tox]
minversion = 1.6
skipdist = True
# 要测试的 Python 版本或环境
envlist = py34,py27,py35,functional,py34-functional,py35-functional,pep8

[testenv]
# 安装依赖
deps = -r{toxinidir}/requirements.txt
      -r{toxinidir}/test-requirements.txt
install_command = pip install -U {opts} {packages}
usedevelop = True
setenv = VIRTUAL_ENV={envdir}
        OS_TEST_PATH=ceilometer/tests/unit
passenv = OS_TEST_TIMEOUT OS_STDOUT_CAPTURE OS_STDERR_CAPTURE \
          OS_LOG_CAPTURE
# 测试时要执行的命令
commands =
    {toxinidir}/tools/pretty_tox.sh "{posargs}"
    oslo-config-generator --config-file=
        etc/ceilometer/ceilometer-config-generator.conf
whitelist_externals = bash
```

对于我们的开发,通常只需要运行下面的两个 tox 命令。

```
$ tox -e pep8      //代码规范检查
$ tox -e py27
```

第一次执行时,会自动安装一些依赖的软件包,如果自动安装失败,我们需要根据提示

信息手动进行安装。

如果我们只希望执行特定的单元测试代码，不喜欢浪费时间去等待所有单元测试的执行，可以加参数指定，比如为了执行 `ceilometer/tests/compute/virt/xenapi/test_inspector.py`：

```
$ tox -e py27 -- test_inspector
```

2.5.4 持续集成 Jenkins

根据《重构—改善代码既有的设计》作者 Martin Fowler 大师在《持续集成》一书中的定义，“持续集成是一种软件开发实践。在持续集成中，团队成员频繁集成他们的工作成果，一般每人每天至少集成一次，也可以多次。每次集成会经过自动构建（包括自动测试）的检验，以尽快发现集成错误。许多团队发现这种方法可以显著减少集成引起的问题，并可以加快团队合作软件开发的速率。”

通俗地说，持续集成（CI）需要对每一次代码提交走一次从代码集成到打包发布的完成流程，以判断提交的代码会对这整个流程带来什么影响。而这个过程中所使用的手法严重依赖于团队成员的多少、目标平台和配置的不同等因素。比如，只有一个人单干，同时只面向一个平台，那么每次有一个 `commit` 时，手工跑一下测试基本就知道结果，也就完全不需要其他更为复杂的工具与手段。

但是对于 OpenStack 这样的项目来说，显然没有这么简单，涉及一个使用版本控制软件来维护的代码仓库，自动的构建过程，包括自动编译、测试、部署等，以及一个持续集成服务器。

1. Jenkins

OpenStack 使用 Jenkins 搭建自己的持续集成服务器。对于一般的小团队开发来说，通常可能会先 `commit` 然后再跑 CI，而 OpenStack 则不同，它通过 Gerrit 首先对每个 `commit` 进行 `review`，这个时候，Jenkins 会执行整个 CI 的过程，通过会“+1”，否则标记“-1”，如图 2-11 所示。

图 2-11 所示的 Jenkins 栏表明了 Jenkins 针对这个 `commit` 都进行哪些测试以及结果，这个列表以 `gate-ceilometer-`为前缀的是 `tox` 的测试结果，比如 `gate-ceilometer-pep8` 对应的即是运行“`tox -e pep8`”。

Author	Qiaowei Ren <qiaowei.ren@intel.com>	Aug 6, 2014 5:04 PM	Code-Review	+2 Nadya Shakhat gordon chung
Committer	Qiaowei Ren <qiaowei.ren@intel.com>	Aug 27, 2014 11:02 AM	Verified	+1 Igor Degtiarov
Commit	2fc9301e154a08a94f71d4254061bd3681deec68	■ (gitweb)	Workflow	+2 Jenkins
Parent(s)	9af6af5e52f8d66b0fd3f6674aae71607b14626d	■ (gitweb)		+1 Nadya Shakhat
Change-Id	I1f6e33696770ec2f4e5cbb2e54e29991978aed32	■		
			Jenkins (2 rechecks)	Aug 28, 2014
			gate-ceilometer-docs	SUCCESS in 5m 26s
			gate-ceilometer-pep8	SUCCESS in 5m 05s
			gate-ceilometer-python26	SUCCESS in 5m 26s
			gate-ceilometer-python27	SUCCESS in 4m 09s
			gate-ceilometer-python33	SUCCESS in 7m 34s
			gate-tempest-dsvm-full	SUCCESS in 45m 46s
			gate-tempest-dsvm-postgres-full	SUCCESS in 1h 00m 04s
			gate-tempest-dsvm-neutron-full	SUCCESS in 47m 38s
			gate-tempest-dsvm-neutron-heat-slow	SUCCESS in 24m 23s
			gate-grenade-dsvm	SUCCESS in 38m 38s
			gate-devstack-dsvm-cells	SUCCESS in 14m 35s
			gate-swift-dsvm-functional	SUCCESS in 17m 48s

图 2-11 Jenkins 结果

2. Tempest

根据上文的描述，Jenkins 背后需要依托大量的单元测试以及集成测试代码，单元测试的代码位于各个项目自身的源码树里，而 OpenStack 的集成测试则是使用 Tempest 作为框架。

Tempest 是 OpenStack 项目中一个独立的项目，它的源码位于 `/opt/stack/tempest/`（使用 Devstack 部署开发环境），包含了大量的测试用例。

```
$ cd /opt/stack/tempest
$ tree -d -L 2
├── data
├── doc
├── source
├── etc
├── releasenotes
├── notes
├── source
├── tempest
├── api
├── api_schema
├── cmd
├── common
├── hacking
├── lib
├── scenario
├── services
├── stress
├── test_discover
├── tests
└── tools

$ cd tempest/services/
```

```
$ tree -d -L 1
├── baremetal
├── data_processing
├── identity
├── object_storage
├── orchestration
└── volume
```

etc/目录下是 Tempest 的配置文件，tools/目录是一些辅助脚本，所有的测试用例都在 tempest/目录。

tempest.api 主要测试 OpenStack API 部分的功能，tempest.cli 测试 OpenStack CLI 接口，tempest.scenario 主要根据一些复杂场景进行测试，包括启动 VM、挂载 volume 和网络配置等，tempest.stress 压力测试，tempest.services 则是自己实现的 API 客户端，是对各个项目 API 的封装，目的是不让一些 bug 隐藏在官方实现的客户端里面。

以 tempest.api 为例，它里面的所有测试用例都是基于 tempest.test.TestCase，这个基类声明了 setUpClass 方法，在类初始化的时候调用。tempest.api.<project>.base 中对这个基类进行了继承，比如 tempest.api.compute.base.BaseV2ComputeTest，里面又实现了很多工具函数，供各项目 API 相关的测试用例调用。有了这些工具函数，具体的测试用例就可以很方便地编写。

比如 tempest.api.compute.flavors.test_flavors.FlavorsV2TestJSON，继承自 BaseV2ComputeTest，所以在初始化的时候，就会把 Tempest 自己实现的 API Client 赋值给类的属性，然后在具体的测试函数里，FlavorsV2TestJSON 就利用这个 Client 的函数来对 OpenStack 进行查询。

```
@test.idempotent_id('6e85fde4-b3cd-4137-ab72-ed5f418e8c24')
def test_list_flavors_with_detail(self):
    # Detailed list of all flavors should contain the expected flavor
    flavors = self.client.list_flavors(detail=True)['flavors']
    flavor = self.client.show_flavor(self.flavor_ref)['flavor']
    self.assertIn(flavor, flavors)
```

测试用例 test_list_flavors_with_detail 就是首先利用 flavor client 来获取所有的 flavor 列表，再获取具体的某个 flavor，然后验证这个 flavor 的确是在所有的 flavor 里面的。

而这里使用的就是 Tempest 自己实现的 RESTful API Client，具体实现位于 tempest.services。

我们提交代码到 Gerrit 上后，Jenkins 会执行包括集成测试在内的各项测试，但有时候仍然需要我们在本地执行集成测试，比如针对新功能的 patch 引发 Tempest 某些测试用例执行失败，需要我们首先修改 Tempest 代码（通常做法是注释掉这个失败的测试用例，并将修改提交给 Tempest，等 Tempest 接受后，原来的 patch 集成测试会成功通过，等到它们被相应的项目接受后再来修改 Tempest 代码并提交）。

本地执行 Tempest 测试可以使用 nose 或者 testr 工具。

执行所有 Tempest 测试用例，代码如下：

```
$ nosetests tempest
$ testr run --parallel
```

执行某一个测试用例，代码如下：

```
$ nosetests tempest.api.compute.flavors.test_flavors.py: \
FlavorsV2TestJSON.test_list_flavors
$ testr run -parallel tempest.api.compute.flavors. \
test_flavors.py: FlavorsV2TestJSON.test_list_flavors
```

3. 第三方 CI

如前所述，Jenkins 是 OpenStack 的官方 CI 系统，每一个 patch 在最终合并前都必须通过 Jenkins 的测试。

此外，还有许多第三方提供的自动化测试系统用于帮助验证、测试特定的 patch，通称第三方 CI，比如图 2-12 所示的 Reviewers 一栏里的 Intel NFV CI、Intel PCI CI、Mellanox CI 等。

Reviewers	Citrix XenServer CI Intel NFV CI Jenkins XenProject CI Dan Smith IBM PowerKVM CI Intel PCI CI Jay Pipes John Garbutt Mellanox CI Microsoft Hyper-V CI Moshe Levi Paul Carlton Quobyte CI Sergey Nikitin VMware NSX CI Virtuozzo CI Virtuozzo Storage CI melanie witt
Project	openstack/nova
Branch	master
Topic	bug/1512880
Strategy	Merge if Necessary
Updated	14 hours ago

Code-Review	
Verified	+1 Citrix XenServer CI Intel NFV CI Jenkins XenProject CI
Workflow	
Jenkins check	Aug 25 11:23 PM
gate-nova-docs-ubuntu-xenial	SUCCESS in 3m 55s
gate-nova-pep8-ubuntu-xenial	SUCCESS in 8m 42s
gate-nova-python27-db-ubuntu-xenial	SUCCESS in 13m 38s
gate-nova-python34-db	SUCCESS in 10m 57s
gate-nova-python35-db-nv	SUCCESS in 11m 41s (non-voting)

图 2-12 第三方 CI

第三方 CI 也是通过 Gerrit 系统接入 Openstack 的开发流程。每提交一个 patch, Gerrit 会发布一个事件, Jenkins 监听 Gerrit 事件, 启动 patch 测试或者 Gate (将代码合并入主干) 流程。每一个第三方 CI 一般都只关注某个官方项目, 测试专门一部分代码, 比如 Intel PCI Test 只接受 Nova Patch Set Create 事件, 启动针对 PCI 子系统的测试, 测试结果通过 Gerrit 反馈给 Openstack 社区, 最终在该 patch 相应 Gerrit 评审页面的 Reviewer 一栏就能看到 Intel PCI Test 的测试结果。

第三方的 CI 系统基本上都会基于成熟的 Jenkins 测试系统, 如图 2-13 所示, 最基本的配置包括一个 Jenkins Master、几个测试端、一个用于发布测试日志的开放的 Web/FTP 服务器, 测试日志要保留至少几个月时间。Openstack Infra 对 Jenkins 和 Web Server 的设置都有具体的规定和指导 (参见 http://docs.openstack.org/infra/system-config/third_party.html)。

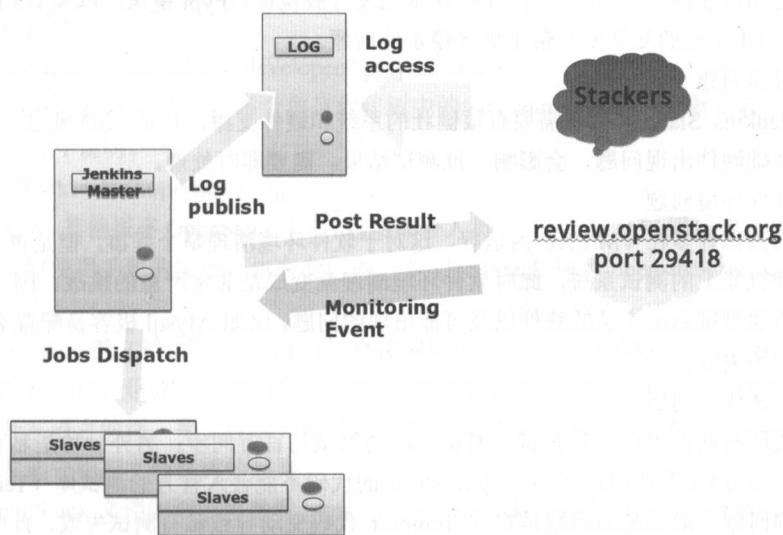


图 2-13 第三方 CI 基本架构

Jenkins 官方提供了 Gerrit trigger 插件, 连接到 `review.openstack.org:29418` 接收官方 Gerrit 事件, 在 trigger 插件内过滤出感兴趣的变化, 触发 Jenkins 具体的测试, 比如 Intel PCI Test 会在 Nova Patch 提交的时候触发针对 PCI 的测试。

当测试完成, 需要将测试日志发布到公开的 Log Server 内, 并根据测试将结果反馈给 Gerrit, 同时将日志连接一并发回给 Gerrit。之后开发者 (Stackers) 就能看到测试结果, 访问测试日志。

根据 OpenStack 官方要求, 每个第三方 CI 都需要申请一个专用 OpenStack 账号, 用于接入官方 CI 系统。第三方 CI 的申请人需要确保第三方 CI 反馈给社区有意义的结果, 并保证 7 × 24 小时运行, 对于出现的问题要及时处理, 确保 OpenStack Infra 团队可以联系到维护人员。

最重要的要求有以下几个：

1) 及时处理问题，更新 CI 状态。(https://wiki.openstack.org/wiki/ThirdPartySystems)

2) 积极参与 Infra team IRC 会议。

(https://wiki.openstack.org/wiki/Meetings/InfraTeamMeeting)

3) 及时处理 CI 出现的各种问题。

第三方 CI 最容易出现如下的一些问题：

(1) 网络问题

OpenStack 测试环境一般是基于 Devstack，整个测试环境从开始搭建到结束耗时为 20~50 分钟，时间的长短与测试的复杂程度相关。其中需要大量访问 Internet，包括发行版、Python 库和大量 Git 仓库。任意时刻出现网络不稳定或断网，都会导致测试失败，或用时增加。

为了提高可靠性，许多第三方 CI 搭建本地发行版镜像、Pypi 镜像，以及 Git 镜像，这无疑又增加了 CI 系统的复杂度，带来更多稳定性问题。

(2) 机器问题

作为 Jenkins Slave 的机器需要有很健壮的系统 and 硬件支撑，包括 SSD 硬盘、RAID 系统等。一旦基础硬件出现问题，会影响一批测试结果，需要即时处理。

(3) 软件环境问题

流行的测试许多在 VM/LXC 内运行，这对于软件环境清理是个好事，但是也有需要直接运行在物理机器上的测试系统，此时软件环境清理就变成是非常严重的挑战。因为 Devstack 并不负责清理测试后所安装的软件以及可能出现的问题，比如 Mysql 极易配置/测试不当导致下一次部署失败。

(4) 测试代码问题

测试代码有两大类，一种是官方测试，此类测试与官方同步；另外一种是有私有测试。后一种也是第三方 CI 存在的意义。如果私有测试的代码不能合入官方的测试库 (Tempest)，会导致一系列问题。最常见的问题是官方 Tempest 代码变动导致私有测试失败，此时应该极力将测试代码从 Tempest 中解耦合。

(5) 自动化维护和恢复问题

为了保证迅速处理所发生的问题，自动化的维护部署工具必不可少，基于流行的 Ansible 或者 Puppet 都是不错的选择。

(6) 监控和告警系统

Zabbix 可以提供基本的系统告警，但是还有更详细的告警需求，比如测试持续失败、某台机器测试连续失败、本地镜像不能访问等。

2.6 如何贡献

我们相信，每一个走向软件开发这条“不归路”的开发者都不会仅仅满足于只是在这个

领域打个酱油，都会希望在 community 里发出自己的声音，但是，作为一个弥漫着现实主义的开源社区，你能够发出声音的大小，要取决于你的贡献大小。

2.6.1 文档

如前所述，完善的文档、文档与代码保持同步是影响软件可维护性的一个重要因素，而开发者大多又是勤于写代码懒于写文档的，但是为 OpenStack 完善文档却也不失为提升自己贡献度的一种方式。

OpenStack 的文档主要分为以下 3 类。

(1) Wiki, wiki.openstack.org

OpenStack Wiki 内容包罗万象，既有一些 HowTo 指南，也有很多设计的细节，以及很多会议的信息和记录。

(2) RST, docs.openstack.org/developer/<projectname>

开发者创建的 RST 文件位于各个项目代码树中的/doc/source/目录，也就是我们所谓的开发文档，比如 API 说明。

(3) DocBook, docs.openstack.org

除了开发者，其他很多 OpenStack 用户，包括部署人员、管理人员、API 用户等还用 Docbook 创建了很多类似使用手册的文档，也有专门的一个 OpenStack 项目 `openstack-manuals` 来负责相关的维护工作。

OpenStack 的 3 类文档，除了 Wiki 可以直接编辑外，RST 和 DocBook 文档的创建过程与代码提交的过程类似，区别只是编写的格式以及工具不同而已。如果我们把文档的格式要求也当成一种开发语言的话，那么区别只是代码开发使用的是 Python，要遵守 Python 的规则，而编写文档要遵守相应文档格式的规则。

有关 OpenStack 文档的细节可参看 <https://wiki.openstack.org/wiki/Documentation/HowTo>。

2.6.2 修补 bug

进入一个新的领域，常规的切入手段都是从简单的修修补补开始。作为新的 OpenStack 开发者，通常也是从“fix bug”开始。在这个“fix bug”的过程中，对 OpenStack 的开发流程，代码的提交与管理过程有一个更为深入的理解。

1. 寻找 bug

当然，修补 bug 的前提是要找到一个有眼缘或者适合的 bug，有两种手段可以被采用：一是在使用 OpenStack 的过程中被动等待 bug 出现，二是主动出击在现有已经提交的那些 bug 中发现挑选。

OpenStack 所有已经提交的 bug 都被列在 <https://bugs.launchpad.net/> 供人认领。这些 bug

已经针对项目进行了归类，比如 Nova 的 bug 都在 <https://bugs.launchpad.net/nova>，如图 2-14 所示。

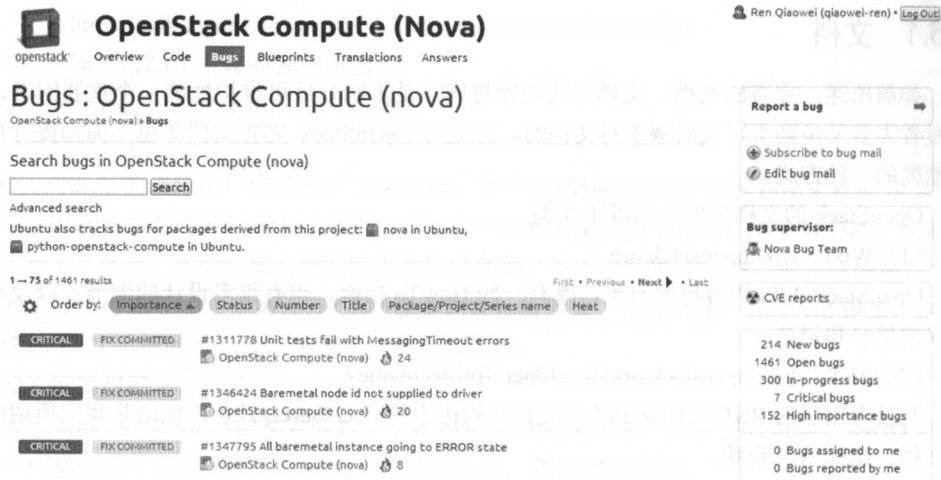


图 2-14 Nova bug 列表

在如图 2-14 所示的页面中，列出了目前已提交的所有 bug，我们可以点击右上角的“Report a bug”提交一个新的自己发现的 bug，也可以打开 bug 列表中的任一 bug 进行查看。

提交新 bug 的过程并不复杂，主要就是对 bug 进行表述并确认是否有类似的 bug，困难的是如何去发现一个新的 bug，因而对 OpenStack 新人来说，更为简单明了的方式就是从 bug 列表中找一个来认领。

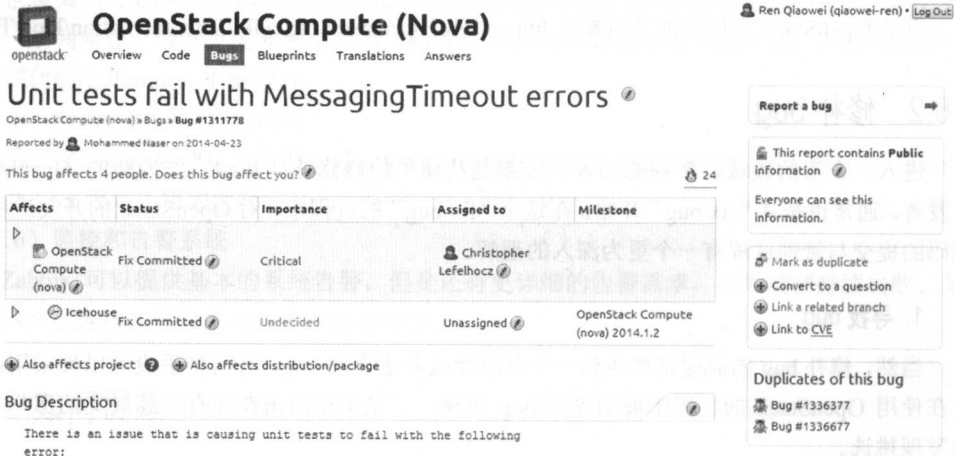


图 2-15 具体 bug 的页面

如图 2-15 所示为具体某一个 bug 的详细描述页面，显示有这个 bug 的优先级以及当前状态等信息，在“Assigned to”中会显示是否分配出去或者已经被人认领。如果它显示的内容是“Unassigned”，也就是说无主状态，我们就可以结合一下 bug 的内容和自身的条件进行一番评估，合适的话就可以分配给自己进行认领。

2. 提交 patch

认领一个合眼缘的 bug 只是第一步，接下来我们需要去 fix 它，然后把我们的代码提交到 Gerrit 供人 review。

我们不能直接在 master 分支上进行修改，而是要创建一个专门的分支来针对这个 bug 进行修补，标准的流程应该如下：

```
$ cd nova
$ git checkout -b bug1335559
$ git commit -a
$ git review -t bug/1335559
```

这里的“1335559”是每个 bug 专属的 id，可以在 bug 的详细描述页面上看到。需要注意的是，当我们使用“git commit”提交代码的时候，不要忘记在描述信息里加上“Closes-Bug: #1335559”。

通过“git review”命令将我们的 patch 成功提交到 Gerrit 之后，就可以在 <https://review.openstack.org> 上打开该 bug 相应的页面查看当前 review 的过程，并与其他开发者针对我们的修改进行互动。

2.6.3 增加 feature

通俗的理解，bug 是毛病，feature 是功能，一个惹人厌，一个高大上。其实，有时候 feature 只是穿了衣服的 bug，如图 2-16 所示。

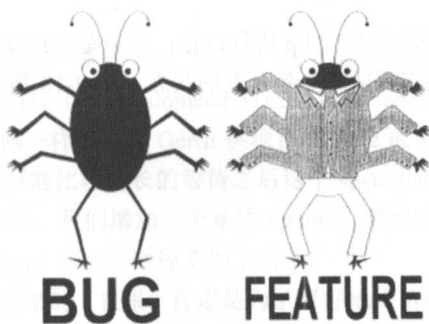


图 2-16 bug 与 feature 的区别

因此，与 fix bug 的流程相比，本节的重点就在于多出的那层“衣服”。

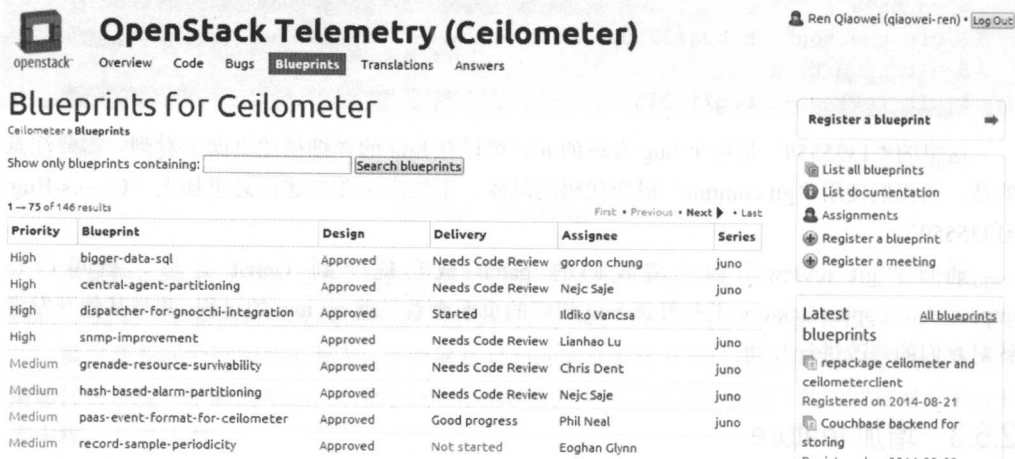
1. bp (blueprint)

与修改 bug 相比，增加 feature 时穿上的那件“衣服”就是 bp。

简单来说，bp 主要阐述了有关新 feature 的一些想法与设计，以及该 bp 的 milestone，可以用于跟踪相关开发人员的开发进程。对于一些复杂的 feature，还会准备有相关的 wiki 页面。

与 bug 类似，所有已经创建的 bp 都被列在 <https://blueprints.launchpad.net/>，也可以去各个项目专属的页面查看相关的 bp，比如 <https://blueprints.launchpad.net/ceilometer>。

在如图 2-17 所示的页面中，列出了目前 Ceilometer 项目已创建的所有 bp，我们可以单击右上角的“Register a blueprint”创建一个新的 bp，也可以打开 bp 列表中的任一 bp 进行查看。



OpenStack Telemetry (Ceilometer)

Overview Code Bugs **Blueprints** Translations Answers

Blueprints for Ceilometer

Ceilometer » Blueprints

Show only blueprints containing: [Search blueprints](#)

1 — 75 of 146 results

Priority	Blueprint	Design	Delivery	Assignee	Series
High	bigger-data-sql	Approved	Needs Code Review	gordon chung	juno
High	central-agent-partitioning	Approved	Needs Code Review	Nejc Saje	juno
High	dispatcher-for-gnocchi-integration	Approved	Started	Ildiko Vancsa	
High	snmp-improvement	Approved	Needs Code Review	Lianhao Lu	juno
Medium	grenade-resource-survivability	Approved	Needs Code Review	Chris Dent	juno
Medium	hash-based-alarm-partitioning	Approved	Needs Code Review	Nejc Saje	juno
Medium	paas-event-format-for-ceilometer	Approved	Good progress	Phil Neal	juno
Medium	record-sample-periodicity	Approved	Not started	Eoghan Glynn	

First • Previous • Next • Last

Register a blueprint

- List all blueprints
- List documentation
- Assignments
- Register a blueprint
- Register a meeting

Latest blueprints [All blueprints](#)

- repackage ceilometer and ceilometerclient
Registered on 2014-08-21
- Couchbase backend for storing

图 2-17 Ceilometer bp 列表

创建 bp 的过程同样并不复杂，主要就是填写一个合适的标题并对 feature 进行表述，困难的是创建之后能够被接受。

项目的 core 团队会对所有创建的 bp 进行讨论，决定是否接受以及它的优先级。在 bp 被接受后，开发过程中我们还需要适时更新开发的状态。图 2-18 所示为具体 bp 的详细描述页面。

Add support to xenapi in ceilometer

Ceilometer » Blueprints » Add support to xenapi in ceilometer

Registered by jiang, yunhong on 2013-01-06

currently ceilometer only support libvirt hypervisor, we need add support to xenapi also.

Read the full specification

Blueprint information

Status:

Started

Priority:

Low

Direction:

Approved

Definition:

Approved

Milestone target:

juno-3

Approver:

Nick Barcet

Drafter:

jiang, yunhong

Assignee:

Ren Qiaowei

Series goal:

Accepted for juno

Implementation:

Good progress

Started by

gordon chung on 2014-08-07

Completed by

Related branches

Link a related branch

Related bugs

Link a bug report

Sprints

Propose for sprint

Whiteboard

Gerrit topic: <https://review.openstack.org/#q:topic:xenapi-support,n,z>

Addressed by: <https://review.openstack.org/102529>

Spec for xenapi support

图 2-18 bp 的详细描述页面

2. spec

在 spec 出现之前, 我们增加新 feature 时, 只需要给它穿一件很简洁的“衣服”, 创建一个 bp 等待讨论接受。我们需要花费大量精力的地方在于如何让该 feature 相关的 patch 被 merge 到项目中。

但是, 在 spec 出现之后, 需要穿的“衣服”就复杂了很多。各个项目逐渐又多了一个 <project>-specs 这样伴生项目, 比如 Ceilometer 对应的 ceilometer-specs, 我们需要在里面创建一个 spec, 然后像提交代码一样提交给 Gerrit 供项目的 Core 成员以及其他开发者 review, 在经过若干次的 update 和有可能比较漫长的等待之后这个 spec 可能会被接受。

也就是说, 在 spec 之前, 我们增加一个新的 feature, 需要经历一个 Gerrit 评审的过程; 在 spec 出现之后, 这个 Gerrit 评审的过程变为了两个。

存在即合理, 这个多出来的“衣服”肯定是为了更好地规范项目的开发。每个 specs 项目都会包含一个模板文件, 新创建的每个 spec 必须按照这个模板逐项填写, 包括相应的 bp 链接、问题的描述、对 Rest API 等可能的影响、实现的设计细节以及参考资料等内容。基本上填完

内容，实现的各种细节已经了然于胸，只剩代码了。

而原本的 bp 不需要考虑这么复杂，我们可以看到很多被接受的 bp 也仅仅寥寥几句，只是描述了一下想法而已。

3. 提交 patch

除了上述要穿上的衣服 bp 和 spec，与修改 bug 相比，增加新 feature 需要提交 patch 的过程大体相同，即把完成的代码提交到 Gerrit 供大家 review。

我们同样不能直接在 master 分支上进行实现，而是要创建一个专门的分支来针对这个 feature，标准的流程应该如下：

```
$ cd ceilometer
$ git checkout -b xenapi-support
$ git commit -a
$ git review -t bp/xenapi-support
```

这里的“xenapi-support”是我们创建 bp 时指定的标题。当我们使用“git commit”提交代码的时候，不要忘记在描述信息里加上“Implements: blueprint xenapi-support”。

通过“git review”命令将我们的 patch 成功提交到 Gerrit 之后，就可以在 <https://review.openstack.org> 上打开该 bp 相应 patch 的页面查看当前 review 的过程，并与其他开发者针对我们的实现进行互动。

2.6.4 review

除了贡献文档、代码之外，review 其他开发者的 patch 是我们向 OpenStack 的另外一种重要方式。

图 2-19 所示为 30 天内 Nova 项目的贡献排名 (<http://stackalytics.com/report/contribution/nova/30>)，除了提交的 patch 数目之外，很重要的一部分就是 review 的数目。

“Engineer”栏人名后面加“*”表示 Core Developer，有“+2”与“-2”的权限。如果你想加入 Core 团队，在这个排名里要尽量靠前是必要条件。

而 review 本身只是要求我们花费一定的时间去浏览理解其他开发者的代码，有针对性地提出自己的问题并做出“+1”或“-1”的评价。

如图 2-20 所示，在代码上双击即可出现输入框，输入我们的疑问。

Contribution into nova for the last 30 days

Search:

#	Engineer	Reviews	-2	-1	+1	+2	A	+	%	Disagreements	Ratio	On review / patch sets	Commits	Emails
1	Mikhail Durnosvistov (Mirantis)	264	0	137	127	0	0	48.1%		18	6.8%	1 / 13	0	0
2	Jay Pipes (Mirantis) *	227	1	58	24	144	43	74.0%		17	7.5%	7 / 29	7	18
3	Daniel Berrange (Red Hat) *	216	4	76	7	129	54	63.0%		17	7.9%	16 / 90	20	30
4	Gary Kotton (VMware) * stable/havana	176	0	70	100	6	4	60.2%		15	8.5%	21 / 112	21	12
5	Dan Smith (Red Hat) *	171	5	73	2	91	39	54.4%		8	4.7%	20 / 92	23	7
6	Ken'ichi Ohmichi (NEC) *	137	0	49	1	87	30	64.2%		6	4.4%	30 / 112	11	0
7	Russell Bryant (Red Hat) *	122	2	28	0	92	51	75.4%		3	2.5%	12 / 67	10	31
8	Matt Riedemann (IBM) *	103	2	44	1	56	38	55.3%		0	0.0%	26 / 73	20	13
9	Sylvain Bauza (Red Hat)	98	0	43	55	0	0	56.1%		6	6.1%	4 / 19	1	18
10	Ihar Hrachyshka (Red Hat) * stable/havana	86	12	15	4	55	35	68.6%		3	3.5%	4 / 11	3	2
11	Olav Vix (IBM)	85	0	43	47	0	0	49.4%		4	4.7%	20 /	11	5

图 2-19 30 天内 Nova 贡献排名

```
def _lookup_by_name(self, instance_name):
    vm_refs = self._call_xenapi("VM.get_by_name_label", instance_name)
    n = len(vm_refs)
    if n == 0:
        raise virt_inspector.InstanceNotFoundException(
            (Draft)
            
            Save Discard
            _('VM %s not found in XenServer') % instance_name)
```

图 2-20 review 代码

2.6.5 调试

这个世界从来就不缺少文艺青年，所以即使在 IT 博客论坛抑或书店的 IT 专柜，我们都经常看到“编码的艺术”“调试的艺术”以及“注释的艺术”等糅合了屌丝气息和人文情怀的字眼。而与另外两种“艺术”强调个体创造相比，“调试的艺术”则需要在充分的和机器的不断交流中完成，也因此一些具有浪漫主义情怀的 IT 文艺男青年会说“我喜爱调试代码胜过了写代码”。

但是本书写不出这样的人文关怀，也相信看到这些文字的读者都已经写过调试过很多的代码，即使不是使用 Python 语言，也调试过 C 代码，对调试目的以及包括断点在内的一些基

本概念也都了然。所以这里只简单介绍一种最为常用的调试 OpenStack 代码,也就是调试 Python 代码的手段——PDB。

使用 PDB 调试 OpenStack 代码只需要在我们希望设置断点的地方加上下面的两行代码:

```
import pdb
pdb.set_trace()
```

然后在 OpenStack 各个服务所在的那个 Screen 里重启相应的服务,代码就会停止在这两行代码所在的地方,然后我们可以使用与 GDB 类似的一些命令进行调试,比如使用“p”打印一些信息,使用“n”单步调试等。

云计算的一个核心思想就是在服务器端提供集中的物理计算资源。这些计算资源可以被分解成更小的单位去独立地服务于不同的用户，也就是在共享物理资源的同时，为每个用户提供隔离、安全、可信的虚拟工作环境，而这一切不可避免地要依赖于虚拟化技术。

3.1 概述

对于虚拟化，每个人都可能会有自己的认识。但其实所谓的虚拟化技术已经存在40多年。比如，在计算机发展的“上古时代”，曾有一段时间开发者会担心是否有足够的可用内存来存放自己的程序指令和数据，于是这时操作系统里引入了虚拟内存的概念，这是操作系统为了满足应用程序的需求，对内存进行的虚拟和扩展。

再比如，因为购买大型机系统价格十分昂贵，系统管理员又不希望各部门的用户独占资源，所以出现了所谓的虚拟服务器，能够让用户更好地时分共享（Time-sharing）昂贵的大型机系统。

当然，虚拟化技术的内涵远远不止于虚拟内存和虚拟服务器这么简单。如果我们在一个更广泛的环境中或从更高级的抽象，比如任务负载虚拟化和信息虚拟化，来思考虚拟化技术，虚拟化技术就变成了一个非常强大的概念，可以为最终用户、上层应用和企业带来很多好处。

现代计算机系统是一个庞大的整体，整个系统的复杂性是不言而喻的。因此，计算机系统自下而上地被分成了多个层次。图3-1所示为一种常见的层次划分。

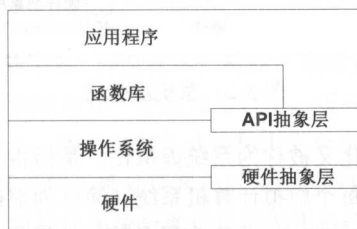


图3-1 计算机系统层次结构

每一个层次都向上一层次呈现一个抽象，并且每一层只需要知道下层抽象的接口，而并不需要了解其内部运作机制。比如，操作系统所看到的硬件是一个硬件抽象层，它并不需要理解硬件的布线或者电气特性。

这样层次抽象的好处是，每一层只需要考虑本层的设计以及与相邻层间的交互接口，从而大大降低了系统设计的复杂性，提高了软件的移植性。从另一个方面来说，这样的设计也是给下一层软件模块为上一层软件模块创造“虚拟世界”提供了条件。

本质上，虚拟化就是由位于下层的软件模块，根据上一层软件模块的期待，抽象出一个虚拟的软件或硬件接口，使上一层软件可以直接运行在与自己所期待的运行环境完全一致的虚拟环境上。

虚拟化可以发生图 3-1 所示的各个层次上，不同层次的虚拟化会带来不同的虚拟化概念。在学术界和工业界里，也先后出现了各种形形色色的虚拟化概念，这也是我们前面为什么会说“对于虚拟化，每个人都可能会有自己的认识”。有人认为虚拟内存和虚拟服务器都是虚拟化，有人认为硬件抽象上的虚拟化是一种虚拟化，也有人认为类似 Java 虚拟机这种软件也算是一种虚拟化。

对云计算而言，特别是提供基础架构即服务的云计算，更关心的是硬件抽象层上的虚拟化。因为，只有把物理计算机系统虚拟化为多台虚拟计算机系统，通过网络将这些虚拟计算机系统互联互通，才能够形成现代意义上的基础架构，即服务云计算系统。

如图 3-2 所示，硬件抽象层上的虚拟化是指通过虚拟硬件抽象层来实现虚拟机，为客户机操作系统呈现出与物理硬件相同或相近的硬件抽象层。由于客户机操作系统所能看到的只是硬件抽象层，因此客户机操作系统的行为和其在物理平台上没有什么区别。

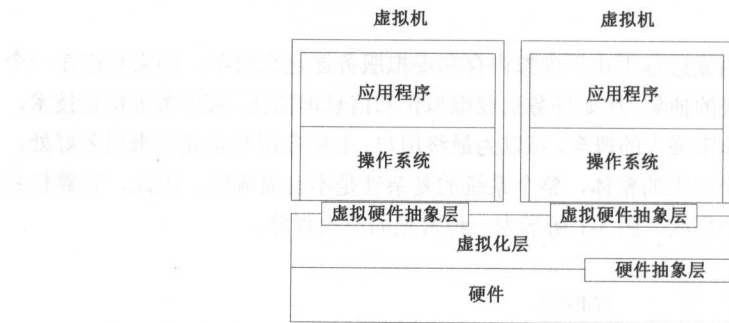


图 3-2 系统虚拟化

这种硬件抽象层上的虚拟化又被称为系统虚拟化，是指将一台物理计算机系统虚拟化为一台或多台虚拟计算机系统。每个虚拟计算机系统（简称为虚拟机，也就是上面所称的客户机）都拥有自己的虚拟硬件，如 CPU、内存和设备等，并提供一个独立的虚拟机执行环境。通过虚拟机监控器（Virtual Machine Monitor, VMM，也可以称为 Hypervisor）的模拟，虚拟机中的操作系统（Guest OS，客户机操作系统）认为自己仍然是独占一个系统在运行。在一台物理计算机上运行的每个虚拟机中的操作系统都是可以完全不同的，并且它们的执行环境是完全独立的。

3.1.1 虚拟化实现方式

当前主流的虚拟化按照实现方式可以分为两种：

- VMM 直接运行在硬件平台上，控制所有硬件并管理客户操作系统。客户操作系统运行在比 VMM 更高的级别。这个模型也是虚拟化历史里的经典模型，很多著名虚拟机都是根据这个模型来实现的，比如说 Xen。
- VMM 运行在一个传统的操作系统里（第一软件层），可以看做是第二软件层，而客户机操作系统则是第三软件层了。KVM 和 VirtualBox 就是这种实现。

两种实现方式的具体区别如图 3-3 所示。

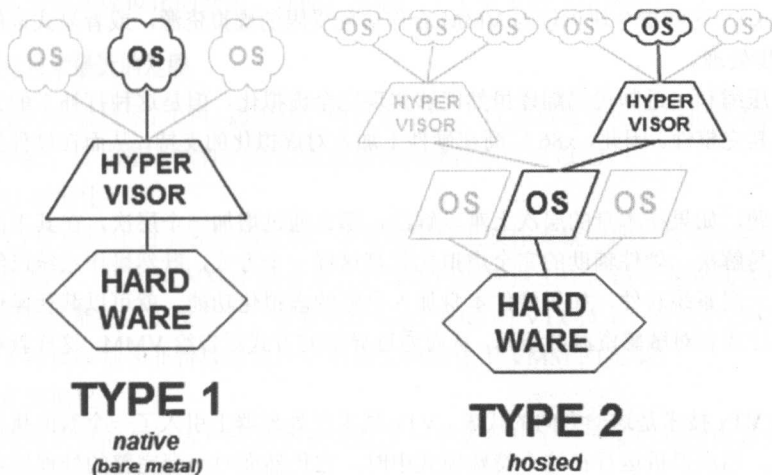


图 3-3 虚拟化的不同实现方式

按照 VMM 所提供的虚拟平台类型又可以将 VMM 分为两类。

(1) 完全虚拟化 (Full Virtualization)

VMM 虚拟的是现实存在的平台，并且在客户机操作系统看来，虚拟平台和现实平台是一样的，客户机操作系统感觉不到运行在一个虚拟平台上，现有的操作系统无需进行任何修改就可以运行在这样的虚拟平台上，因此这种方式被称为完全虚拟化。

完全虚拟化中，VMM 需要能够正确处理客户机操作系统所有可能的行为，或者说正确处理所有可能的指令，因为客户机操作系统会像正常的操作系统那样，去操作虚拟处理器、虚拟内存和虚拟外设。从实现方式来说，完全虚拟化经历了两个阶段：软件辅助的完全虚拟化与硬件辅助的完全虚拟化。

在 x86 虚拟化技术的早期，x86 体系没有在硬件层次上对虚拟化提供支持，因此完全虚拟化只能通过软件来实现。一个典型的做法是优先级压缩 (Ring Compression) 和二进制代码翻译 (Binary Translation) 相结合。

优先级压缩的原理是：将 VMM 和客户机的优先级放到同一个 CPU 中来运行，对应于 x86 架构，通常是 VMM 在 ring 0，客户机操作系统内核在 ring 1，客户机操作系统应用程序在 ring 3。当客户机操作系统内核执行特权指令时，由于处在非特权的 ring 1，通常会触发异常，VMM 截获后就可以进行特权指令的虚拟化。但是 x86 指令体系在设计之初并没有考虑到虚拟化，一小部分特权指令在 ring 1 中没有触发异常，VMM 也就不能截获进行虚拟化。所以这些特权指令不能通过优先级压缩来进行虚拟化。

因此二进制代码翻译被引入来处理这些虚拟化不友好的指令。就是通过扫描并修改客户机的二进制代码，将难以虚拟化的指令转化为支持虚拟化的指令。VMM 通常会对操作系统的二进制代码进行扫描，一旦发现虚拟化不友好的指令，就将其替换成支持虚拟化的指令块（Cache Block）。这些指令块可以与 VMM 合作访问受限的虚拟资源，或者显式地触发异常让 VMM 进一步处理。

优先级压缩和二进制代码翻译虽然能够实现完全虚拟化，但是这种打补丁的方式很难在构架架上保证其完整性。因此，x86 厂商在硬件上加入对虚拟化的支持，从而在硬件架构上实现了虚拟化。

很多问题，如果在本身的层次上难以解决，那么通过增加一个层次，在其下面一个层次就会变得容易解决。硬件辅助的完全虚拟化就是这样一个方式，既然操作系统已经是硬件之上的最下面一层系统软件，如果硬件本身加入足够的虚拟化功能，就可以截获操作系统对敏感指令的执行或者对敏感资源的访问，从而通过异常的方式报告给 VMM，这样就解决了虚拟化的问题。

Intel 的 VTx 技术是这一方向的代表。VTx 技术在处理器上引入了一个新的执行模式用于运行虚拟机。当虚拟机运行在这个特殊模式中时，它仍然面对一套完整的处理器寄存器和执行环境，只是任何特权操作都会被处理器截获并报告给 VMM。VMM 本身运行在正常模式下，在接收到处理器的报告后，通过对目标指令的解码，找到相应的虚拟化模块进行模拟，并把最终的效果反映在特殊模式的环境中。

硬件虚拟化是一种完备的虚拟化方法，因为内存和外设的访问本身也是由指令来承载的，对处理器指令级别的截获就意味着 VMM 可以模拟一个与真实主机完全一样的环境。在这个环境中，任何操作系统只要能够在现实中的等同主机上运行，也就可以在这个虚拟机环境中无缝运行。

（2）类虚拟化（Para-Virtualization）

第二类 VMM 虚拟出的平台是现实中不存在的，而是经过 VMM 重新定义的。这样的虚拟平台需要对所运行的客户机操作系统进行或多或少的修改使之适应虚拟环境，客户机操作系统也就知道自己运行在虚拟平台上，并且会去主动适应。这种方式被称为类虚拟化。

类虚拟化是通过在源代码级别修改指令以回避虚拟化漏洞的方式，来使 VMM 能够对物理资源实现虚拟化。对于 x86 中难以虚拟化的指令，完全虚拟化通过 Binary Translation 在二进制代码级别上来避免虚拟化漏洞。类虚拟化采取的是另一种思路，即修改操作系统内核的

代码,使得操作系统内核完全避免这些难以虚拟化的指令。既然内核代码已经需要修改,类虚拟化可以进一步优化 I/O。也就是说,类虚拟化不是去模拟真实世界中的设备,因为太多的寄存器模拟会降低性能。相反,类虚拟化可以自定义出高度优化的 I/O 协议。这种 I/O 协议完全基于事务,可以达到近似物理机的性能。

3.1.2 虚拟化现状和未来

虚拟化技术自从 20 世纪 60 年代诞生以来,一直在飞速发展。尤其是近年来,IT 管理技术和云计算的大规模应用对虚拟化技术提出了更高的要求,也促使硬件厂商、软件提供商使用更新的技术来提高虚拟化的安全、性能,从而产生了更多的应用场景。

1. 虚拟化技术最近的发展

虚拟化中的核心技术:CPU 虚拟化、内存虚拟化、IO 虚拟化和网络虚拟化都经历了一个前面提到的革新,即由基于软件的虚拟化全面转向硬件辅助虚拟化。

(1) CPU 虚拟化

早先的 CPU 虚拟化由于硬件的限制,必须将客户机操作系统中的特权指令替换成可以陷入到 VMM 的指令,从而让 VMM 接管并且进行相应的模拟工作,最后返回到客户机操作系统中。这种做法性能差、工作量大、容易引起 bug。Intel 的 VTx 技术,对现有的 CPU 进行了扩展,引入了特权级别和非特权级别,从而极大地简化了 VMM 的实现。

(2) 内存虚拟化

内存虚拟化的目的是给客户机操作系统提供一个从零开始的连续的物理内存空间,并在各个虚拟机之间进行有效的隔离。

客户机物理地址空间并不是真正的物理地址空间,它和宿主机物理地址空间还有一层映射关系。内存虚拟化要通过两次地址转换来实现,即 GVA (Guest Virtual Address, 客户机虚拟地址)到 GPA (Guest Physical Address, 客户机物理地址)再到 HPA (Host Physical Address, 宿主机物理地址的转换)。

其中 GVA 到 GPA 的转换是由客户机软件决定的,通常由客户机看到的 CR3 指向的页表来指定;GPA 到 HPA 的转换则由 VMM 决定。VMM 在将物理内存分配给客户机时就确定了 GPA 到 HPA 的转换,并将这个映射关系记录到内部数据结构中。

原有的 x86 架构只支持一次地址转换,即通过 CR3 指定的页表来实现“虚拟地址”到“物理地址”的转换,这无法满足虚拟化两次地址转换的要求。因此原先的内存虚拟化就必须将两次转换合并为一次转换来解决这个问题,即 VMM 根据 GVA 到 GPA 再到 HPA 的映射关系,计算出 GVA 到 HPA 的映射关系,并将其写入所谓的“影子页表”(Shadow Page Table)。影子页表尽管实现了传统的内存虚拟化,但是实现非常复杂,内存开销很大,性能也会受到影响。

为了解决“影子页表”的局限,Intel 的 VTx 提供了 EPT (Extended Page Table) 技术,

直接在硬件上支持 GVA/GPA/HPA 的两次地址转换，大大降低了内存虚拟化的难度，提高了相关性能。此外，为了进一步提高 TLB 的使用效率，VTx 还引入了 VPID (Virtual Processor ID) 技术，进一步优化内存虚拟化的性能。

(3) I/O 虚拟化

传统的 I/O 虚拟化方法主要有“设备模拟”和“类虚拟化”。前者通用性强，但性能不理想；后者性能不错，却又缺乏通用性。如果要兼顾通用性和高性能，最好的方法就是让客户机直接使用真实的硬件设备。这样客户机的 I/O 操作路径几乎和没有虚拟机的环境下相同，从而可以获得几乎同样的性能。因为这些是真实存在的设备，客户机可以使用自带的驱动程序去发现并使用它们，通用性的问题也得以解决。但是客户机直接操作硬件设备需要解决如下两个问题：

- 如何让客户机直接访问到设备真实的 I/O 地址空间（包括 I/O 端口和 MMIO）。
- 如何让设备的 DMA 操作直接访问到客户机的内存空间。因为设备不管当前运行的是虚拟机还是真实操作系统，都会用驱动提供给它的物理地址做 DMA 操作。

VTx 技术已经能够解决第一个问题，允许客户机直接访问物理的 I/O 空间。Intel 的 VTd 技术则是为了解决第二个问题，它提供了 DMA 重映射 (Remapping) 技术，以帮助 VMM 的实现者达到目标。

VTd 技术通过在北桥引入 DMA 重映射硬件，以提供设备重映射和设备直接分配的功能。在启用 VTd 的平台上，设备所有的 DMA 传输都会被 DMA 重映射硬件截获，然后根据设备对应的 I/O 页表，对 DMA 中的地址进行转换，使设备只能访问限定的内存。这样，设备就可以直接分配给客户机使用，驱动提供给设备的 GPA 经过重映射，变为 HPA，使得 DMA 操作得以完成。

(4) 网络虚拟化

早期的网络虚拟化都是通过重新配置宿主机的网络拓扑结构来实现的，比如将宿主机的网络接口和代表客户机的网络接口配置在一个桥接 (Bridge) 下面，使得客户机可以拥有独立的 Mac 地址，并且在网络中就像一个真正的物理机一样。但是这种方法加大了宿主机网络驱动负担，降低了系统性能。

VTd 技术可以将一个网卡直接分配给客户机使用，从而达到和物理机一样的性能。但是它的可扩展性又比较差，因为一个物理网卡只能分配给一个客户机，而且服务器能够支持的最多 PCI 设备数总是有限的，远远不能满足越来越多的客户机数量。因此 SRIOV (Single Root I/O Virtualization) 被引入，来解决这个问题。

SRIOV 是 PCIe (PCI Express) 规范的一个扩展，定义了本质上可以共享的新型设备。它允许一个 PCIe 设备，通常是网卡，为每个与其连接的客户机复制一份资源（例如内存空间、中断和 DMA 数据流），使得数据处理可以不再依赖 VMM。SRIOV 定义了两种 function 的类别。

- PF (Physical Function): 完整的 PCIe function, 定义了 SRIOV 的能力, 用于配置和管理 SRIOV。
- VF (Virtual Function): 轻量级的 PCIe function, 只包括进行数据处理 (Data Movement) 的必要资源, 和 PF 或其他 VF 共享另外的物理资源, 可以看做是设备的一个虚拟化的实例。

在虚拟化的环境下, 一个 VF 被当做一个虚拟网卡分配给客户机操作系统, 所有的 VF 和 PF 被连接在 SRIOV 网卡内部的一个桥 (bridge), 这样各个 VF 的通信可以互不干扰, 网络数据流也绕开了原先的 VMM 中的软件交换机实现, 并且直接在 VF 和客户机操作系统间传递。因此, 消除了原来的软件模拟层, 达到了几乎和非虚拟化环境一样的网络性能。具体的使用可以参考图 3-4。

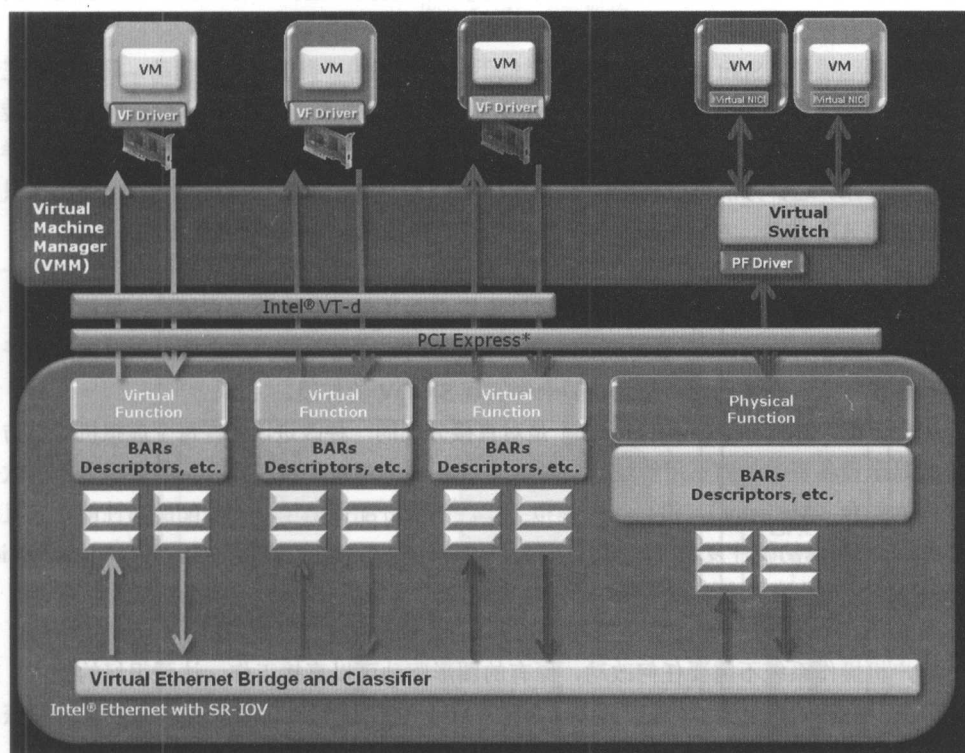


图 3-4 VTd 的实现

(5) GPU 虚拟化

图 3-5 所示为 GPU 虚拟化的常用方法。

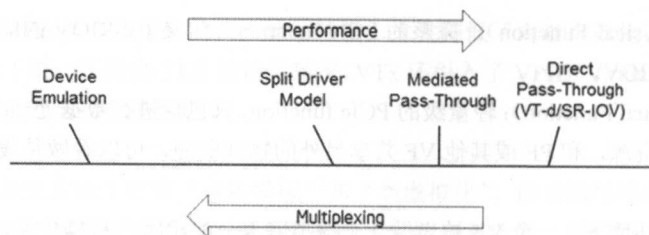


图 3-5 GPU 虚拟化常用方法

设备模拟（Device Emulation）是最传统的方法，QEMU 就是典型代表。它模拟了一个比较简单的 VGA 设备模型，截获客户机操作系统对 VGA 设备的操作，然后利用宿主机上的图形库绘制最终的显示结果。绘制哪怕最简单的图形，也要经过多次客户机、宿主机之间的通信，而且没有硬件帮助进行加速，所以性能很差。

分离驱动模型（Split Driver Model）类似于前面提到的“类虚拟化”驱动，只不过它工作在 API 的层面。前端驱动（Front End Driver）将客户机操作系统的 DirectX/OpenGL 调用转发到宿主机的后端驱动（Back End Driver）。后端驱动就像一个在宿主机上运行的 3D 程序一样，进行绘制工作。这种方法可以利用宿主机的 DirectX/OpenGL 库实现硬件加速，但是由于只能针对特定的 API 加速，对宿主机、客户机和运行其中的 3D 程序都有各种限制。

直接分配（Direct Pass-Through）基于前面提到的 VTd 或者 SRIOV 等技术，直接将一个硬件分配给客户机操作系统来用。VTd 可以将整个 PCI 显卡分配给客户机用，性能很好，但是可扩展性较差。SRIOV 标准使得 PCI 设备本质上可以在各个客户机之间共享，但是由于显示硬件过于复杂，众多厂商不愿意在显卡中实现 SRIOV 的扩展。

中介分配（Mediated Pass-Through）是对直接分配的一种改进，允许每个虚拟机可以访问部分的显示设备资源，而不用经过 VMM 的任何干涉。但对于特权操作，需要引入新的软件模块作为中介（Mediator），进行相关模拟工作。中介分配保留了直接分配的高性能，并且避免了 SRIOV 实现的硬件复杂性，是比较成熟的解决方案。Intel 的 GVT-g (Graphics Virtualization Technology) 就是这种方法的典型代表。

- Intel XenGT: XenGT 是由 Intel GVT-g 的 Xen 实现，架构如图 3-6 所示。

客户机操作系统不需要任何改动，原有的图形驱动可以直接工作，达到很好的性能。对于部分关乎性能的重要资源，客户机可以不经 VMM 而直接访问。但特权操作会被 Xen 截获，并且转发到中介（Mediator）。中介为每个客户机创建一个虚拟的 GPU 上下文（Context），并在其中模拟特权操作。当 VM 发生切换时，中介也会切换 GPU 上下文。XenGT 将中介的实现放在 dom0 中，这样就可以避免在 VMM 里面增加复杂的设备逻辑，而且减轻了发布时的工作。

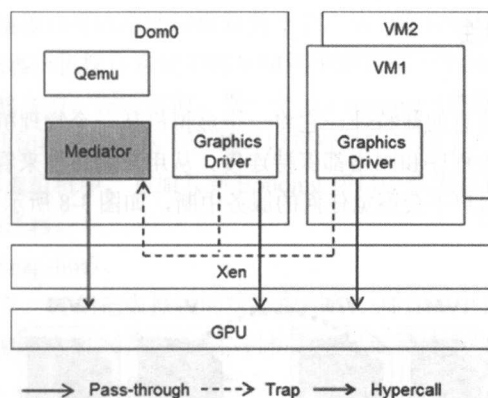


图 3-6 XenGT 架构

XenGT 目前已经开始了对于 Xen 的集成, 相信不久的将来, Xen 的用户就可以享受到一个客户机进行 3D 运算, 另一个客户机运行 3D 游戏的乐趣。

- Intel KVMGT: KVMGT 是由 Intel GVT-g 的 KVM 实现, KVMGT 只支持 Intel 的 GPU, 从 Haswell 开始支持。架构如图 3-7 所示。

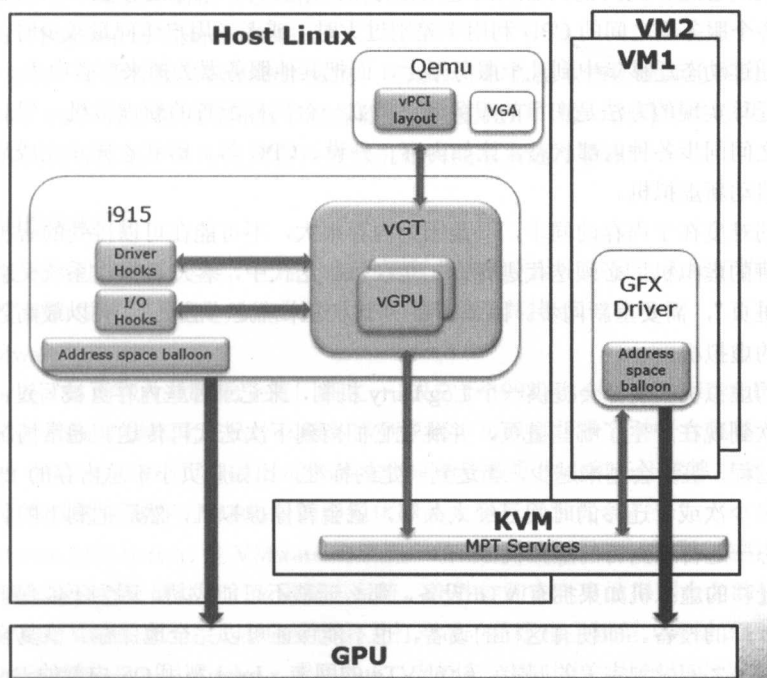


图 3-7 KVMGT 架构

2. 虚拟化引入的新特性

(1) 动态迁移

动态迁移是虚拟化特有的新特性，它将一个虚拟机从一个物理机快速迁移到另外一个物理机，但是虚拟机里面的程序和网络都保持连接。从用户的角度来看，动态迁移对虚拟机的可用性没有任何影响，用户不会察觉任何的服务中断，如图 3-8 所示。

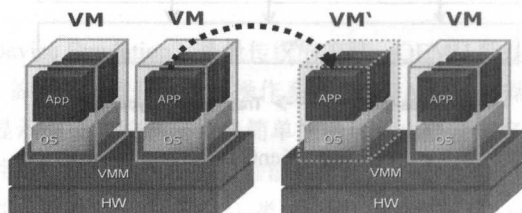


图 3-8 动态迁移

动态迁移最大的好处就是提高服务器的可维护性。当察觉到即将发生的硬件故障时，可以把虚拟机动态迁移到其他机器从而避免服务中断。另外，动态迁移也可以用于负载均衡。比如，当各个服务器之间的 CPU 利用率差别过大时，或者当用户访问量较少时，可以将所有的虚拟机通过动态迁移集中到几个服务器上，而把其他服务器关掉来节省电力。

动态迁移实现的方法是在目的服务器上建立一台同样配置的新虚拟机，然后不断地在两个虚拟机之间同步各种内部状态，比如内存、外设、CPU 等。等状态同步完成后，关掉老的虚拟机，启动新虚拟机。

实现的难度在于内存的同步，一是因为内存很大，不可能在可以接受的宕机时间内一下子同步到目的虚拟机，必须迭代进行。二是在每次迭代中，客户机操作系统又会写内存，造成新的“脏页”，需要重新同步。因此需要一个方法来确定“脏页”，并以最高效的迭代算法同步到目的虚拟机。

常见的虚拟机实现都会提供一个 Log Dirty 机制，来记录哪些内存页被写过。每次迭代都会查看上次到现在产生了哪些脏页，并跳过它们留到下次迭代再传送。通常情况下，这是一个收敛的过程，脏页会越来越少，当达到一定的标准，比如脏页小于总内存的 1%，迭代次数已经超过多少次或者迁移的时间已经太久等，就会暂停虚拟机，然后把剩下的脏页和虚拟机的其他状态一起传送到目的虚拟机。

动态迁移的虚拟机如果拥有 VTd 设备，那么迁移不可能成功。因为不能保证目的物理机也有完全一样的设备。即使有这样的设备，也不能保证可以完全地保存、恢复设备状态，从而在两个设备之间做到完美的同步。针对 VTd 的网卡，Intel 利用 OS 内部的 Bonding Driver 和 Hotplug 机制，提供了一套软件的解决方案。

Bonding Driver 可以将多个网卡绑定成一个网络接口，提供一些高级功能，比如热备份，

当一个网卡失效了，网络接口可以自动切换到另一个，从而保证网络连接的通畅。VTd 网卡的动态迁移，就是事先在客户机操作系统中将 VTd 网卡绑定在一个热备份的 Bonding 接口下，而用一个虚拟网卡作为热备份。在迁移前，做一个 hot remove（热插拔）的操作将 VTd 网卡移除，Bonding 接口自动切换到虚拟网卡，就可以进行动态迁移了。迁移成功后，再 hot add 一个 VTd 的网卡到目的虚拟机中，并加入到 Bonding 接口中作为默认的网卡。这样就巧妙地实现了 VTd 网卡的动态迁移。

（2）虚拟机快照（Snapshot）

虚拟机快照就是在某一时刻把虚拟机的状态像照片一样保存下来。通常，快照要保存所有的硬盘信息、内存信息和 CPU 信息。虚拟机快照可以便捷地产生一套同样的虚拟机环境，因此被广泛地用于测试、备份和安全等各种场景。

（3）虚拟机克隆

虚拟机克隆是指把一个虚拟机的状态完全不变地复制到另外一个虚拟机，形成两个完全相同的系统，并且可以同时运行。为了达到同时运行的目的，新的虚拟机的某些配置，比如 Mac 地址，可能需要改动以避免和老虚拟机的冲突。

如今的数据中心都由数以万计的机器组成，部署工作也需要耗费大量的时间和精力。有了虚拟机克隆技术，只需要先安装配置好一台虚拟机，然后克隆到其他数以万计的虚拟机中，从而大大降低了整个数据中心的安装和配置时间。

（4）P2V（Physical to Virtual Machine）

P2V 是指将一个物理服务器的操作系统、应用程序和数据从物理硬盘迁移到一个虚拟机的硬盘镜像中。P2V 技术极大地降低了服务器虚拟化的使用门槛，使得用户可以方便地将现有的物理机转化成虚拟机，从而使用各种虚拟机相关技术来进行管理。

3. 典型的虚拟化产品

虚拟化技术经过多年的发展，已经出现了很多成熟的产品，应用也从最初的服务器扩展到了桌面等更宽的领域。这里是几种典型的虚拟化产品及其特点。

（1）VMware

VMware 是 x86 虚拟化软件的主流厂商之一，成立于 1998 年，并于 2003 年被 EMC 收购。VMware 提供一系列的虚拟化产品，从服务器到桌面，可以运行于各种平台包括 Windows、Linux 和 Mac OS。近年来，VMware 的产品线也延伸到数据中心和云计算等方面，形成了各个层次、各个领域的全覆盖。VMware 的虚拟化产品主要有以下几个。

- VMware ESX Server: 是 VMware 的旗舰产品，基于 Hypervisor 模型（类型 1 VMM），直接运行在物理硬件上，无需操作系统，在性能和安全方面得到全面的优化。
- VMware Workstation: 是面向桌面的主打产品，基于宿主模型（类型 2 VMM），宿主机操作系统可以是 Windows 或 Linux。支持完全虚拟化，因此可以使用各种客户机操作系统，包括 Windows、Linux、Solaris 和 FreeBSD。

- **VMware Fusion:** 也是面向桌面的一款产品，功能和 VMware Workstation 基本相同，但是 Fusion 的宿主机操作系统是 Mac OS X，并且有很多针对 Mac 系统的优化。

VMware 产品具有很多优点：

- 功能丰富。很多新的虚拟化功能都是最先由 VMware 开发的。
- 配置和使用方便。VMware 开发了非常易于使用的配置工具 and 用户界面。
- 稳定，适合企业级应用。VMware 产品非常成熟，很多企业选择 VMware ESX Server 来运行关键应用。

(2) Microsoft

微软在虚拟化方面起步比 VMware 晚，但在认识到虚拟化的重要性之后，微软通过外部收购和内部开发，推出了一系列产品，涵盖了用户状态 (User State) 虚拟化、应用程序 (Applications) 虚拟化和操作系统虚拟化。操作系统虚拟化产品主要有面向桌面的 Virtual PC 和面向服务器的 Virtual Server。这些产品的特点在于和 Windows 操作系统结合得非常好，在 Windows 下非常易于配置和使用。

(3) Xen

Xen 起源于英国剑桥大学的一个研究项目，逐渐发展成一个开源软件项目，吸引了许多公司和科研院所加入，发展非常迅速。

从技术角度来说，Xen 基于混合模型，特权操作系统 (Domain 0 或者说 dom0) 起到了宿主机操作系统的很多管理功能，其他非特权的虚拟机 (domU) 运行用户的程序。Xen 最初是基于类虚拟化来实现的，通过修改 Linux 内核，实现处理器和内存的虚拟化，通过引入 I/O 的前端/后端驱动 (front/backend) 架构实现设备的虚拟化。利用类虚拟化的优势，Xen 可以达到接近物理机的性能。它的构架如图 3-9 所示。

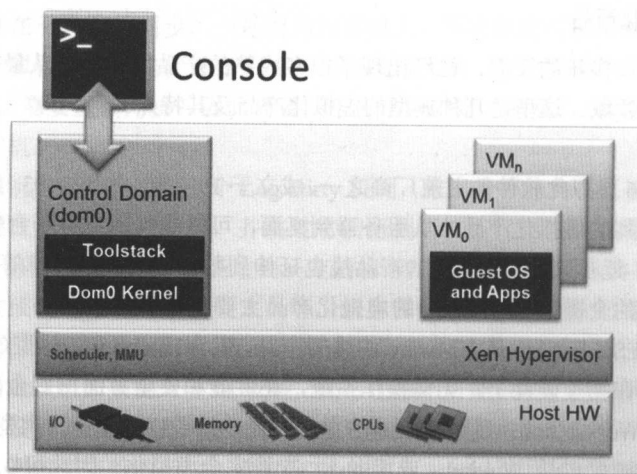


图 3-9 Xen 架构

随着 Xen 社区的发展壮大，硬件完全虚拟化技术也被加入到 Xen 中，比如 Intel VT 和 AMD-V，因此未加修改的操作系统也可以在 Xen 上面运行。

Xen 支持多种硬件平台，官方的版本支持包括 x86_32、x86_64、IA64、PowerPC 和 ARM 架构。Xen 目前已经比较成熟，基于 Xen 的虚拟化产品很多，如 Citrix，VirtualIron，Redhat 和 Novell 等都有相应的产品。

作为开源软件，Xen 的主要特点如下：

- 可移植性非常好，开发者既可以将其移植到其他平台，也可以将其修改用于项目研究。
- 独特的类虚拟化支持，提供了接近于物理机的性能。但 Xen 的易用性和其他成熟的商业产品还有一定的差距，有待加强。

(4) KVM

KVM (Kernel-based Virtual Machine) 也是一款基于 GPL 的开源虚拟机软件。它最早由 Qumranet 公司开发，在 2006 年 10 月出现在 Linux 内核的邮件列表上，并于 2007 年 2 月被集成到了 Linux 2.6.20 内核中，成为内核的一部分。

KVM 的架构如图 3-10 所示。它是基于 Intel VT 等技术的硬件虚拟化，并利用 QEMU 来提供设备虚拟化。此外，Linux 社区中已经发布了 KVM 的类虚拟化扩展。

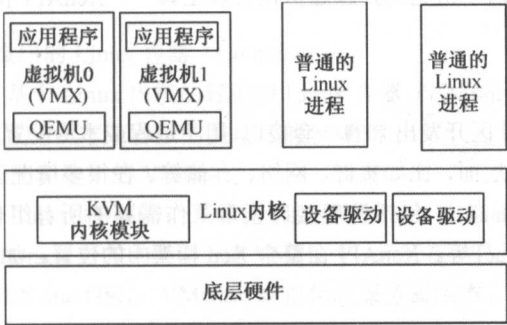


图 3-10 KVM 架构

从架构上看，KVM 属于宿主模型（类型 2），因为 Linux 设计之初并没有针对虚拟化的支持，KVM 是以内核模块的形式存在的。但是随着越来越多的虚拟化功能被加入到 Linux 内核当中，也可以把 Linux 内核看做一个 Hypervisor。因此 KVM 也可以算做 Hypervisor 模型（类型 1）。

3.2 高层管理工具

VMM 本身只是提供了虚拟化的基础构架，很多和最终用户相关的工作，比如配置、启动虚拟机，都要作为高层的管理工具来实现。这些管理工具存在于宿主机中，一般包括各种应

用程序和库文件。

高层管理工具的引入，主要是基于两个原因：

- 分层管理，屏蔽底层细节。最终用户只会关心高层的功能，而 VMM 的实现细节，对于最终用户应该是透明的。因此需要管理工具作为桥梁，接收用户的请求，然后调用 VMM 提供的接口，来完成最终的工作。

比如用户发起请求，要创建一个新的虚拟机。管理工具就会按照严格的时序来完成初始化工作：在宿主机创建设备模型（Device Model），对来自客户机操作系统的 I/O 进行模拟；调用 VMM 提供的接口，为客户机分配内存，和其他所需资源，比如影子页表、计时器（Timer）等；最后分配给虚拟机时间片，让它开始运行。用户只要调用一个管理工具的 API 函数，而不需要了解 VMM 底层的实现，就可以创建虚拟机。

- 屏蔽不同虚拟化实现，提供统一管理接口。如前所述，VMM 的实现多种多样，用户很可能在一个环境中部署不同的 VMM。由于它们的管理库、工具都不相同，给部署、管理增加了很多额外的工作和难度。因此引入高层管理工具，屏蔽各个 VMM 的不同，提供统一的接口给用户就是自然而然的事情了。这样用户的应用程序就可以统一起来，而不必处理各个 VMM 的不同特性。

这里简单介绍一下两个常见的开源虚拟化管理工具——XenAPI 和 Libvirt。

3.2.1 XenAPI

XenAPI 是由 Xen 社区开发出来的一套接口，用于远程或本地配置和管理 Xen 的虚拟机，工作在比 Xen 更高级的层面，比如集群、网络、存储等。在很多情况下，XenAPI 也被看做一整套的工具栈（Tool Stack），包括管理接口正常工作需要的所有组件，比如一个守护进程（Daemon）、xe 命令行工具等。XenAPI 在整个 Xen 构架中的位置，如图 3-11 所示。

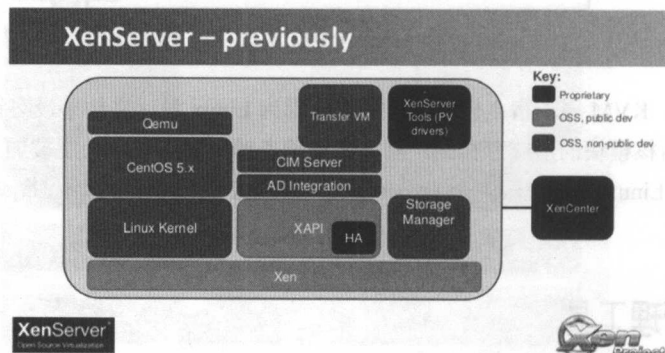


图 3-11 XenAPI 与 Xen

XenAPI 基于 LGPL2 (Lesser GNU General Public License) 发布, 已经进入主流的 Linux 发布版, 像 Ubuntu 和 Debian。XenAPI 伴随着 Xen 项目一起成长, 功能丰富, 性能稳定, 特别适合管理基于 Xen 的虚拟机, 因此是各种云平台, 比如 XCP (Xen Cloud Platform) 和 OpenStack 管理 Xen 的默认配置。

XenAPI 采用 OCaml 语言开发, OCaml 是高性能的面向对象设计语言(性能不亚于 C/C++), 并且支持自动内存管理, 灵活的类型系统。

3.2.2 Libvirt

Libvirt 是由 Redhat 开发的一套开源的软件工具, 目标是提供一个通用和稳定的软件库来高效、安全地管理一个节点上的虚拟机, 并支持远程操作。它由以下模块组成:

- 一个库文件, 实现管理接口。
- 一个守护进程 (libvirtd)。
- 一个命令行工具 (virsh)。

基于可移植性和高可靠性的考虑, Libvirt 采用 C 语言开发, 但是也提供了对其他编程语言的绑定, 包括 Python、Perl、OCaml、Ruby、Java 和 PHP。因此 Libvirt 的调用可以被集成到各种编程语言中, 适应不同的环境。另一方面, 不像 XenAPI 只管理 Xen, Libvirt 支持多种 VMM 具体如下。

- LXC: 轻量级的 Linux 容器 (Container)。
- OpenVZ: 基于 Linux 内核的轻量级 Linux 容器 (Container)。
- KVM/QEMU: 基于 Linux 的类型 2 的 VMM。
- Xen: 开源的类型 1 的 VMM。
- User-mode Linux (UML): 系统调用级别的 Linux 虚拟机。
- VirtualBox: Oracle 开发的类型 2 的 VMM, 可以运行在 Windows, Linux, 和 Mac OS X 上。
- VMware ESX and GSX: VMware 虚拟化的服务器版本。
- VMware Workstation and Player: VMware 虚拟化的桌面版本。
- Hyper-V: 微软开发的 VMM。
- PowerVM: IBM 开发的 VMM, 可以运行在 AIX 和 Linux 上。
- Parallels Workstation: Parallels 为 Mac 开发的 VMM。
- Bhyve: FreeBSD 9+ 上的 VMM。

Libvirt 的层次结构如图 3-12 所示。

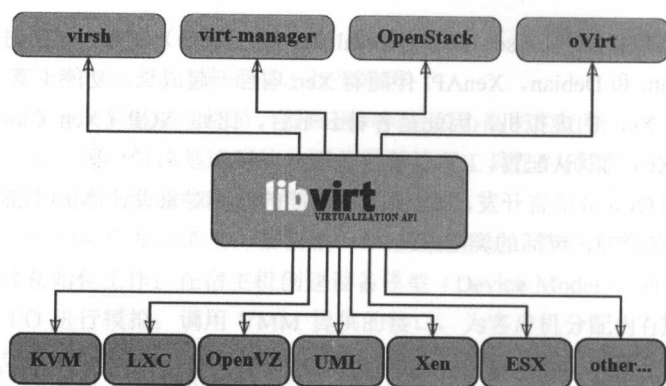


图 3-12 Libvirt 层次结构

为了支持多种 VMM，Libvirt 采用了基于驱动（Driver）的构架，如图 3-13 所示。也就是说，每种 VMM 需要提供一个 Driver，和 Libvirt 进行通信来操控特定的 VMM。这也意味着，通用的 Libvirt 提供的 API 和某种 VMM 可能不完全一样。VMM 的某个接口可能不够通用，因此 Libvirt 并未实现。或者 Libvirt 的某个通用接口在某个 VMM 中并无实际意义，所以也不存在。

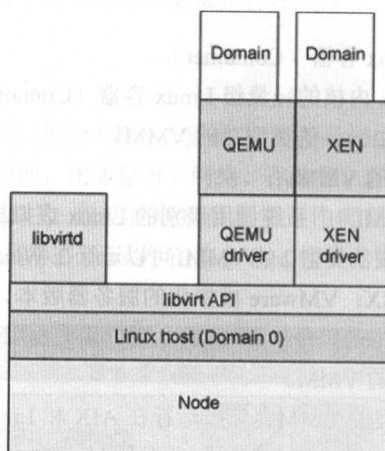


图 3-13 Libvirt 架构

virsh 是一个基于 Libvirt 的命令行工具，用于管理虚拟机的整个生命周期，包括创建、销毁、迁移等。这里是一个简单的创建虚拟机的例子。

在一个安装了 virsh 和 Libvirt 的机器上，首先需要建立一个到特定 VMM 的连接，然后才可以管理这个 VMM。比如使用下面的命令建立一个可以管理 qemu 和 KVM 虚拟机的连接：

```
gzhai@gzhai-cloud:~/aa$ virsh connect qemu:///system
```

然后执行下列命令可以看到宿主机的特性和所支持的客户机：

```
gzhai@gzhai-cloud:~/aa$ virsh capabilities
<capabilities>

<host>
  <uuid>27ef7709-23b8-df11-bbda-6494a61ad485</uuid>
  <cpu>
    <arch>x86_64</arch>
    <model>Nehalem</model>
    <vendor>Intel</vendor>
    <topology sockets='1' cores='4' threads='2' />
    <feature name='rdtscp' />
    <feature name='xtpr' />
    <feature name='tm2' />
    <feature name='est' />
    <feature name='vmx' />
    <feature name='ds_cpl' />
    <feature name='monitor' />
    <feature name='pbe' />
    <feature name='tm' />
    <feature name='ht' />
    <feature name='ss' />
    <feature name='acpi' />
    <feature name='ds' />
    <feature name='vme' />
  </cpu>
  <power_management>
    <suspend_mem />
    <suspend_disk />
    <suspend_hybrid />
  </power_management>
  <migration_features>
    <live />
    <uri_transports>
      <uri_transport>tcp</uri_transport>
    </uri_transports>
  </migration_features>
  <topology>
    <cells num='1'>
      <cell id='0'>
        <cpus num='8'>
          <cpu id='0' />
          <cpu id='1' />
          <cpu id='2' />
```



```

        <cpu id='3' />
        <cpu id='4' />
        <cpu id='5' />
        <cpu id='6' />
        <cpu id='7' />
    </cpus>
</cell>
</cells>
</topology>
<secmodel>
    <model>apparmor</model>
    <doi>0</doi>
</secmodel>
</host>

<guest>
    <os_type>hvm</os_type>
    <arch name='i686'>
        <wordsize>32</wordsize>
        <emulator>/usr/bin/qemu-system-x86_64</emulator>
        <machine>pc-1.0</machine>
        <machine canonical='pc-1.0'>pc</machine>
        <machine>pc-0.14</machine>
        <machine>pc-0.13</machine>
        <machine>pc-0.12</machine>
        <machine>pc-0.11</machine>
        <machine>pc-0.10</machine>
        <machine>isapc</machine>
        <domain type='qemu'>
            </domain>
        <domain type='kvm'>
            <emulator>/usr/bin/kvm</emulator>
            <machine>pc-1.0</machine>
            <machine canonical='pc-1.0'>pc</machine>
            <machine>pc-0.14</machine>
            <machine>pc-0.13</machine>
            <machine>pc-0.12</machine>
            <machine>pc-0.11</machine>
            <machine>pc-0.10</machine>
            <machine>isapc</machine>
        </domain>
    </arch>
</features>
<cpuselection/>

```

```

    <deviceboot/>
    <pae/>
    <nonpae/>
    <acpi default='on' toggle='yes' />
    <apic default='on' toggle='no' />
  </features>
</guest>

<guest>
  <os_type>hvm</os_type>
  <arch name='x86_64'>
    <wordsize>64</wordsize>
    <emulator>/usr/bin/qemu-system-x86_64</emulator>
    <machine>pc-1.0</machine>
    <machine canonical='pc-1.0'>pc</machine>
    <machine>pc-0.14</machine>
    <machine>pc-0.13</machine>
    <machine>pc-0.12</machine>
    <machine>pc-0.11</machine>
    <machine>pc-0.10</machine>
    <machine>isapc</machine>
    <domain type='qemu'>
      </domain>
    <domain type='kvm'>
      <emulator>/usr/bin/kvm</emulator>
      <machine>pc-1.0</machine>
      <machine canonical='pc-1.0'>pc</machine>
      <machine>pc-0.14</machine>
      <machine>pc-0.13</machine>
      <machine>pc-0.12</machine>
      <machine>pc-0.11</machine>
      <machine>pc-0.10</machine>
      <machine>isapc</machine>
    </domain>
  </arch>
  <features>
    <cpuselection/>
    <deviceboot/>
    <acpi default='on' toggle='yes' />
    <apic default='on' toggle='no' />
  </features>
</guest>

</capabilities>

```

Libvirt 使用 XML 格式定义不同的段(section)来显示宿主机和客户机的能力(capability)。这里可以看到宿主机的 CPU 支持 64 位架构, 即 x86_64, 而且支持 vmx 特性, 即硬件级别的虚拟化。支持的客户机有两种: 一种 i686 架构, 一种 x86_64 架构。每种架构的操作系统类型都是 HVM (Hardware Virtual Machine), 就是由硬件支持的虚拟机。域类型 (Domain Type) 指明了支持哪种虚拟机实现, 这里有 QEMU (纯粹的指令模拟器) 和 KVM (开源的虚拟机)。

根据这台机器的情况, 我们可以创建一个 64 位的 KVM 虚拟机。虚拟机的配置也是通过 XML 格式来定义的, 比如下面这个 vml.xml 文件:

```
<domain type='kvm'>
  <name>vml</name>
  <memory>524288</memory>
  <currentMemory>524288</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type arch='x86_64' machine='pc'>hvm</type>
    <boot dev='cdrom'>/>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/kvm</emulator>
    <disk type='file' device='disk'>
      <source file='/home/gzhai/aa/1st.img'>/>
      <target dev='hda' bus='ide'>/>
    </disk>
    <disk type='file' device='cdrom'>
      <source file='/home/gzhai/aa/ubuntu-14.04.1-desktop-amd64.iso'>/>
      <target dev='hdc' bus='ide'>/>
      <readonly/>
    </disk>
    <interface type='network'>
      <source network='default'>/>
    </interface>
    <input type='mouse' bus='ps2'>/>
    <graphics type='vnc' port='-1'>/>
  </devices>
</domain>
```

它定义了一个使用 KVM 的虚拟机，名字叫做 vm1，内存为 512MB（以 KB 为单位），只有一个虚拟 CPU，是 x86_64 架构的 hvm 虚拟机，从光盘启动。特性（features）段里面，定义了 CPU 具有 ACPI（Advanced Configuration and Power Interface）、APIC（Advanced Programmable Interrupt Controller）和 PAE（Physical Address Extension）等功能。设备（devices）段里面，定义了两个磁盘，一个是文件镜像（Image），另一个是 Ubuntu 安装的光盘镜像（ISO）；还定义了一个默认的网络接口，一个 ps2 鼠标。同时指明了使用 vnc 作为虚拟机屏幕渲染的图形库。

有了虚拟机的 XML 配置文件，就可以通过下面的命令创建虚拟机：

```
gzhai@gzhai-cloud:~/aa$ virsh create vm1.xml
Domain vm1 created from vm1.xml
```

通过 vncviewer 可以连接到虚拟机的屏幕，进行从光盘安装操作系统的过程，如图 3-14 所示。

```
gzhai@gzhai-cloud:~/aa$ vncviewer :1
```

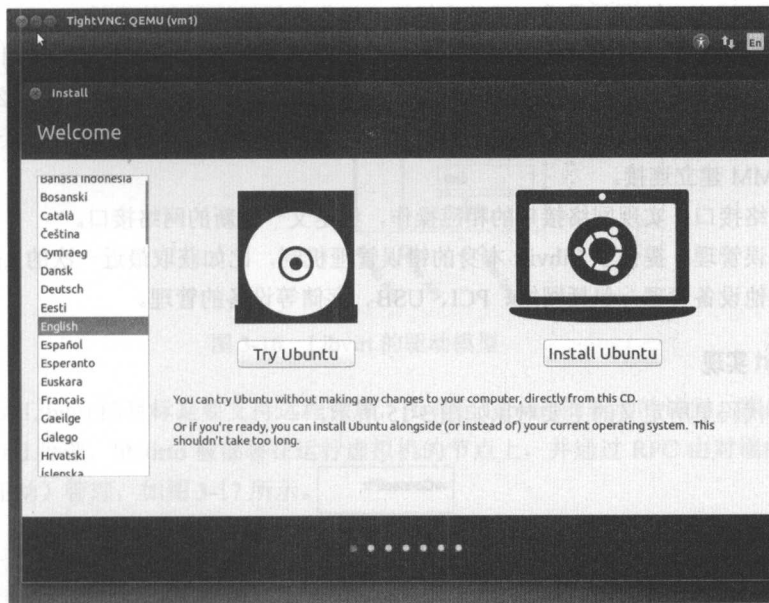


图 3-14 virsh 启动虚拟机安装操作系统

当前虚拟机的运行状态可以通过下面的命令得到：

```
gzhai@gzhai-cloud:~/aa$ virsh list
Id Name          State
-----
7  vm1             running
```

可以看到名字为 vm1，id 为 7 的虚拟机正在运行。

通过下面的命令可以销毁一个虚拟机：

```
gzhai@gzhai-cloud:~/aa$ virsh destroy vm1
Domain vm1 destroyed
```

然后 virsh list 就没有它的信息了：

```
gzhai@gzhai-cloud:~/aa$ virsh list
Id Name
-----
```

1. Libvirt API 简介

Libvirt 定义了各种各样的 API，涉及虚拟化的方方面面，主要分为以下几类。

- 虚拟机快照 (snapshot)：如前所述，快照是包括内存、硬盘灯信息在内的完整虚拟机状态。这些 API 就是用于创建、删除和恢复快照的。
- 虚拟机管理：这一类 API 用于管理虚拟机，也是 Libvirt 里面使用最频繁的功能。比如，创建、销毁、重启、迁移虚拟机，操作虚拟机的磁盘镜像等。
- 事件：事件 (events) 是 Libvirt 定义的一套监测特定情况发生的机制，用户可以通过相应的 API 告诉 Libvirt，想要监测什么样的事件，与事件发生时采取什么样的操作。
- 宿主机：用于获取宿主机的各种信息，包括机器名、CPU 状态等，也用于和特定的 VMM 建立连接。
- 网络接口：实现网络接口的相应操作，如定义一个新的网络接口。
- 错误管理：提供了 Libvirt 本身的错误管理机制，比如获取最近一次的 Libvirt 错误。
- 其他设备管理：包括网络、PCI、USB、存储等设备的管理。

2. Libvirt 实现

Libvirt 代码里所定义的主要对象如图 3-15 所示。

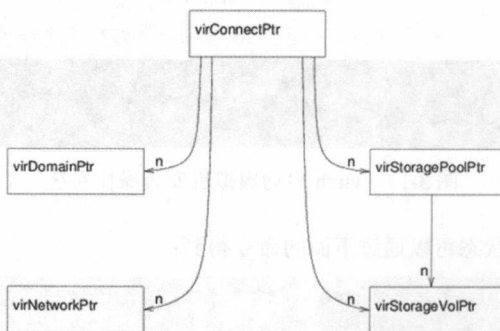


图 3-15 Libvirt 内部的主要对象

- **virConnectPtr**: 代表了与一个特定 VMM 建立的连接。每一个基于 Libvirt 的应用程序都应该先提供一个 URI 来指定本地或远程的某个 VMM，从而获得一个 virConnectPtr 连接。比如 `xen+ssh://host-virt/` 代表了通过 ssh 连接一个在 host-virt 机器上运行的 Xen VMM。拿到 virConnectPtr 连接后，应用程序就可以管理这个 VMM 的虚拟机和对应的虚拟化资源，比如存储和网络。
- **virDomainPtr**: 代表一个虚拟机，可能是激活状态(active)或者仅仅已定义(defined)。已定义表示这个虚拟机存在于固定的配置文件中，可以随时创建一个这样的虚拟机。
- **virNetworkPtr**: 代表一个网络，可能是激活状态或者仅仅已定义。
- **virStorageVolPtr**: 代表一个存储卷，通常被虚拟机当做块设备使用。
- **virStoragePoolPtr**: 代表一个存储池，是用来分配和管理存储卷的逻辑区域。

如前所述，为了支持调用特定的 VMM 功能，Libvirt 使用了驱动(Driver)模型。在初始化过程中，所有的驱动被枚举和注册。每一个驱动都会加载特定的函数为 Libvirt API 所调用。Libvirt 的驱动模型如图 3-16 所示。

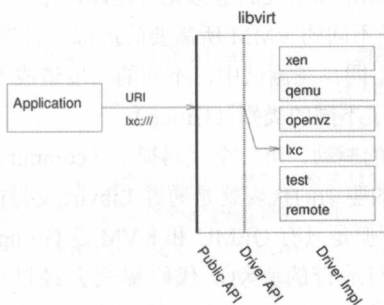


图 3-16 Libvirt 的驱动模型

前面提到，Libvirt 的目标是要支持远程管理，所有到 Libvirt 的驱动访问，都由 Libvirt 守护进程 libvirtd 处理，libvirtd 被部署在运行虚拟机的节点上，并通过 RPC 由对端的 remote Driver（远程驱动）管理，如图 3-17 所示。

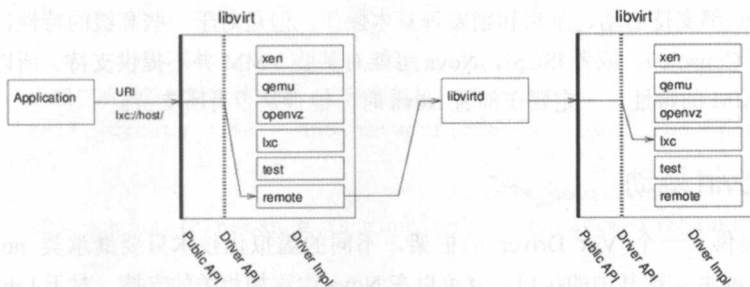


图 3-17 Libvirt 的远程管理模型

如前所述 Libvirt 应用程序用 URI 获得 `virConnectPtr`，代表与一个特定驱动的连接。以后的各种 Libvirt 调用，都要用 `virConnectPtr` 作为参数。在远程管理模式，`virConnectPtr` 实际上连接了本地的 `remote Driver` 和远端的特定 `Driver`。

如图 3-16 所示，所有的调用都通过 `remote Driver` 先到达远端的 `libvirtd`，`libvirtd` 访问对应的 `Driver`（图中是 `lxc`），得到要求的信息和数据并返回给应用程序。应用程序会决定如何处理这些数据，比如进行显示或者写入日志。

3.3 OpenStack 相关实现

OpenStack 中和虚拟化联系最紧密的是 Nova，正是 Nova 使用前面所说的虚拟化管理工具，来管理虚拟机。由于虚拟化管理工具很多，Nova 提供了一个 `Virt Driver` 的框架支持各种的虚拟化实现，也使得用户可以在部署 Nova 时选择使用什么样的管理工具和 VMM（通过在配置文件中选择 `Virt Driver` 来实现）。

部署 Nova，需要选择 VMM 和相关的虚拟化管理库，比如 Xen 既可以通过 Libvirt，也可以通过 XenAPI 来配置。对于不同的 VMM 所需要的虚拟化管理库，Nova 的支持并不相同：在开发中，它们的特性可能不同；在测试中，不同的力度造成不同的成熟度。针对测试和功能，Nova 为 `Virt Driver` 划分了不同的类别（Group）：

- Group A：完全支持的驱动，每一个代码提交（commit）都会经过单元测试和功能测试。这个 Group 内的驱动的代表就是通过 Libvirt 支持的 QEMU 和 KVM（libvirt 还支持其他的 VMM，但是只有 QEMU 和 KVM 是 Group A）。
- Group B：处于中间地带的驱动，代码提交会经过单元测试，但功能测试依赖 OpenStack 以外的系统进行。Group B 中的驱动有 Hyper-v、VMware 和 XenServer（通过 XenAPI）。
- Group C：这个 Group 内的驱动仅有最基本的测试，可能工作也可能不工作。用户使用这些驱动风险自负。代码的提交可能会有单元测试，根本没有公开的功能测试。

Group C 中的驱动有 `baremetal`、`docker`、Xen（通过 Libvirt）和 LXC（通过 Libvirt）。

对于不同的 VMM 和管理库组合，Nova 能够支持的特性也不尽相同。对于绝大多数的 VMM，Nova 都支持启动、重启和销毁等基本操作，但是对于一些高级的特性，比如串口控制台（Serial Console），或者 iSCSI，Nova 可能对某些 VMM 并不提供支持。所以用户如果依赖于某个 VMM 的特性，一定要在部署 Nova 时，检查是否有所支持。

3.3.1 Libvirt 驱动

Nova 提供了一个 `Virt Driver` 的框架，不同的虚拟化技术只要继承类 `nova.virt.driver.ComputeDriver` 并实现其中的接口，就可以在 Nova 中添加相关的支持，对于 Libvirt 来说 `Virt Driver` 的实现就是类 `nova.virt.libvirt.driver.LibvirtDriver`。

Libvirt 提供了很好的 Python 绑定, 所以基于 Python 的 Nova 只要 import 一个名为 libvirt 的 Python 模块, 就可以像 C 程序那样方便地调用 Libvirt 的功能。Libvirt 的 Python 模块也提供了类似 C 库的 Python 类: virConnect 代表一个到 VMM 的连接, virDomain 代表了一个虚拟机。

如前所述, Libvirt 常见的使用模式就是首先建立一个到 VMM 的连接, 即获得一个 virConnect 实例, 有了到 VMM 的连接, 就可以直接调用 virConnect 成员函数来管理虚拟机 (virDomain 实例), 比如 lookupByName 根据虚拟机名字找到并返回一个 virDomain 对象, 然后可以调用 virDomain 的成员函数 info 来得到这个虚拟机的信息。

Nova 与其他基于 Libvirt 库的程序一样, 同样也是这样的使用模式, 并没有什么特别之处。这里以创建虚拟机为例介绍 Nova 与 Libvirt 之间的交互过程。

创建虚拟机时, Nova 最终会调用到类 LibvirtDriver 的 spawn() 函数:

```
def spawn(self, context, instance, image_meta, injected_files,
          admin_password, network_info=None, block_device_info=None):
    # get_disk_info 确定客户机磁盘映射的信息, 包括硬盘和光驱的总线信息, 以及
    # 磁盘映射信息
    disk_info = blockinfo.get_disk_info(CONF.libvirt.virt_type,
                                         instance,
                                         block_device_info,
                                         image_meta)

    # 创建客户机需要的虚拟磁盘镜像, 并把需要的信息写入新的磁盘镜像中
    # 比如网络、磁盘信息、管理员密码和必须的文件
    self._create_image(context, instance,
                       disk_info['mapping'],
                       network_info=network_info,
                       block_device_info=block_device_info,
                       files=injected_files,
                       admin_pass=admin_password)

    # 为新的客户机获取完整配置数据, 包括名称、内存、虚拟 CPU 个数等
    # 然后将配置数据转化成一个 xml 文件, 用于创建虚拟机
    xml = self.to_xml(context, instance, network_info,
                     disk_info, image_meta,
                     block_device_info=block_device_info,
                     write_to_disk=True)

    # 建立所需要的网络并创建、启动虚拟机
    self._create_domain_and_network(context, xml, instance,
                                    network_info,
                                    block_device_info)

    LOG.debug_("Instance is running"), instance=instance)

    def _wait_for_boot():
        state = self.get_info(instance)['state']
```

```

        if state == power_state.RUNNING:
            LOG.info(_("Instance spawned successfully."),
                      instance=instance)
            raise loopingcall.LoopingCallDone()

# 创建一个计时器，每隔 0.5 秒检查一下新建的虚拟机是否启动
# 直到发现它的状态变为正常运行
timer = loopingcall.FixedIntervalLoopingCall(_wait_for_boot)
timer.start(interval=0.5).wait()

```

3.3.2 XenAPI 驱动

Libvirt 的目标是提供通用、稳定的抽象层来安全地管理一个节点上的虚拟机，它也支持 Xen 虚拟机的基本操作，例如创建、销毁，因此似乎 Nova 没有必要再支持 XenAPI。但是 XenAPI 本身又具有自己的一些特性，比如 XenAPI 有“主机池”（pool of hosts）的概念，它是指一些共享存储的宿主机的集合，在主机池中进行动态迁移就没有必要迁移虚拟磁盘。这对 Nova 的实现十分重要，但是 Libvirt 为了通用性的考虑，并不提供这样的特性。为了更好地支持 Xen，Nova 也提供了 XenAPI 的 Virt Driver 支持。

如前所述，Xen 的架构包括 Xen Hypervisor、特权虚拟机 dom0 和非特权虚拟机 domU。基于 Xen 部署 OpenStack 有个特点，就是控制软件（nova-compute）运行在一个 domU 而不是 dom0，这样的好处就是安全地隔离了 dom0 中的系统软件和 OpenStack 组件。图 3-18 所示为一个基于 Xen 的 OpenStack 典型部署，其中包括 dom0、OpenStack domU 和为客户服务的 domU（提供给租户的 Tenant VM）。

- dom0：运行 XenAPI 守护进程 xapi 和一些 OpenStack 组件，比如 xapi 插件和网络隔离规则（Network Isolation Rules）。
- OpenStack domU：在每一个被管理的节点上，都会启动一个 PV domU 运行 OpenStack。它主要启动一个 nova-compute 服务来管理虚拟机，通常也会启动 nova-network 服务来管理可分配给 Tenant VM 的 IP 地址。Nova 通过 XenAPI Virt Driver 使用 XenAPI 的 Python 库和 dom0 中的 xapi 通信来管理虚拟机。dom0 和 domU 间的通信基于宿主的内部网络，因此效率很高。

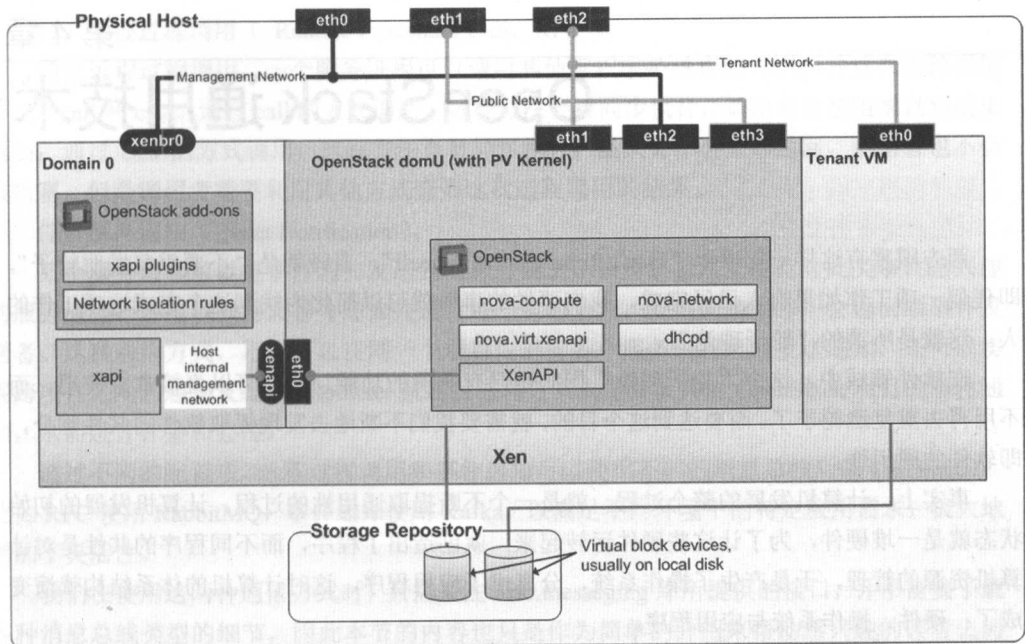


图 3-18 基于 Xen 的 OpenStack 典型部署

OpenStack 通用技术

西方国家有这样一句谚语“Don't Reinvent the Wheel!”, 直译就是“不要重复发明轮子”, 即任何一项工作如果有人已经完成, 我们要做的事情就可以简化为找到这个完成这项工作的人。这就是所谓的“轮子理论”。

在软件领域中, 这个“轮子理论”指的是已有实现的功能, 我们可以直接拿来使用, 而不用再去重复造轮子了。而要达到这个目的, 就需要我们不断地去发现提取软件的共性部分, 即软件的通用性。

事实上, 计算机发展的整个过程, 就是一个不断提取通用性的过程。计算机发展的初始状态就是一堆硬件, 为了让这些硬件运转起来, 就创造出了程序, 而不同程序的共性是对计算机资源的管理, 于是产生了操作系统, 分离出了应用程序, 这时计算机的体系结构就演变成了: 硬件、操作系统与应用程序。

此时所有应用程序的共性变成了对数据的管理, 从而产生了数据库管理系统, 这样一来计算机的体系结构又演变成了: 硬件、操作系统、数据库管理系统与应用程序。

OpenStack 的发展演化过程同样伴随着不断的提取通用性, 从最初的只有 Nova 与 Swift 两个项目, 到目前形形色色的上百个项目, 他们使用的一些通用的技术不断地被提取出来, 由专门的团队 (OpenStack Common Libraries, Oslo) 进行维护。本章将对 OpenStack 通用库, 以及各个项目使用到的大量其他技术和第三方库进行介绍。

4.1 消息总线

OpenStack 遵循这样的设计原则: 项目之间通过 RESTful API 进行通信; 项目内部, 不同服务进程之间的通信, 则必须要通过消息总线。这种设计思想保证了各个项目对外提供服务的接口可以被不同类型的客户端高效支持, 同时也保证了项目内部通信接口的可扩展性和可靠性, 以支持大规模的部署。

软件从最初的面向过程、面向对象, 再到面向服务, 要求我们去考虑各个服务之间如何去传递消息。借鉴硬件总线的概念, 消息总线的模式被引入, 顾名思义, 一些服务向总线发送消息, 其他服务从总线上获取消息。

目前已有多种消息总线的开源实现, OpenStack 也对其中的部分实现有所支持, 比如 RabbitMQ、Qpid 等。基于这些消息总线类型, oslo.messaging 库通过以下两种方式来完成项目内部各服务进程之间的通信。

(1) 远程过程调用 (Remote Procedure Call, RPC)。

通过远程过程调用, 一个服务进程可以调用其他远程服务进程的方法, 并且有两种调用方式: `call` 和 `cast`。通过 `call` 的方式调用, 远程方法会被同步执行, 调用者会被阻塞直到结果返回; 通过 `cast` 的方式调用, 远程方法会被异步执行, 结果并不会立即返回, 调用者也不会被阻塞, 但是调用者需要利用其他方式查询这次远程调用的结果。

(2) 事件通知 (Event Notification)。

某个服务进程可以把事件通知发送到消息总线上, 该消息总线上所有对此类事件感兴趣的服务进程, 都可以获得此事件通知并进行进一步的处理, 处理的结果并不会返回给事件发送者。这种通信方式, 不但可以在同一个项目内部的各个服务进程之间发送通知, 还可以实现跨项目之间的通知发送。`Ceilometer` 就通过这种方式大量获取其他 `OpenStack` 项目的事件通知, 从而进行计量和监控。

通过不同的配置项, 远程过程调用和事件通知可以使用不同的消息总线后端 (backend), 比如 `RPC` 使用 `RabbitMQ`, 事件通知使用 `Kafka`, 以满足不同环境下的特定应用需求, 极大地增加了灵活性。

我们在使用这两种通信方式时, 只需关注 `oslo.messaging` 库所提供的接口, 并不需要了解各种消息总线类型的细节, 因此本节的内容也只是作为简单的介绍来帮助感兴趣的读者去阅读 `oslo.messaging` 的代码。

1. AMQP

`OpenStack` 所支持的消息总线类型中, 大部分都是基于 `AMQP` (`Advanced Message Queuing Protocol`, 高级消息队列协议)。

`AMQP` 是一个异步消息传递所使用的开放的应用层协议规范, 主要包括消息的导向、队列、路由、可靠性和安全性。`Oslo messaging` 中支持的 `AMQP` 主要包括两个版本, `AMQP 0.9.1` 和 `AMQP 1.0`, 这两个版本有着很大的差别, 下面我们以 `RabbitMQ` 主要支持的 `AMQP 0.9.1` 为例介绍一些消息传输中的基本概念。`AMQP` 的架构如图 4-1 所示。

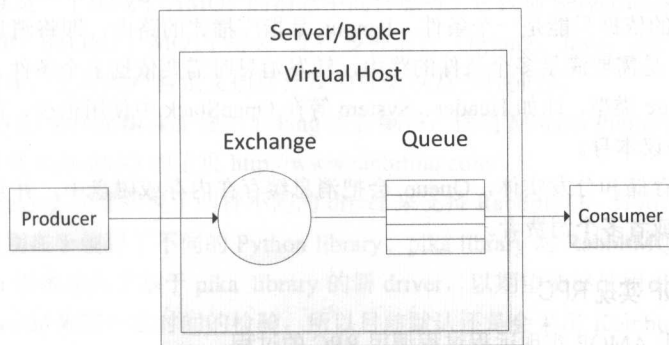


图 4-1 AMQP 架构

对于一个实现了 AMQP 的中间件服务 (Server/Broker) 来说, 当不同的消息由生产者 (Producer) 发送到 Server 时, 它会根据不同的条件把消息传递给不同的消费者 (Consumer); 如果消费者无法接收消息或者接收消息不够快时, 它会把消息缓存在内存或者磁盘上。

AMQP 模型中, 上述操作分别由 Exchange (消息交换) 和 Queue (消息队列) 来实现。此处的虚拟主机 (Virtual Host) 指的是 Exchange 和 Queue 的集合。

生产者将消息发送给 Exchange, 由 Exchange 来决定消息的路由, 即决定要将消息发送给哪个 Queue, 然后消费者从 Queue 中取出消息, 进行处理。

Exchange 本身不会保存消息, 它接收由生产者发送来的消息, 然后根据不同的条件把消息转发到不同的 Queue。这里所谓的“条件”又被称为绑定 (Binding), 可以描述为: 当条件 C 匹配时, 队列 Q 被绑定到交换 E 上。

接收到消息时, Exchange 会查看消息属性、消息头和消息体, 从中提取相关的信息, 然后用此信息再根据绑定表把消息转发给不同的 Queue 或者其他 Exchange。

绝大部分情形下, 这个用来查询绑定表的信息是一个单一的键值, 称为 routing key。每一个发送的消息都有一个 routing key。同样, 每一个 Queue 也有一个 binding key, Exchange 进行消息路由时, 会查询每一个 Queue。如果某个 Queue 的 binding key 与某个消息的 routing key 匹配, 这个消息就会被转发到那个 Queue 中。

不同类型的 Exchange 会使用不同的匹配算法。表 4-1 为 AMQP 中所包含的比较重要的 Exchange 类型。

表 4-1 消息交换类型

类 型	说 明
Direct	binding key 和 routing key 必须完全一致, 不支持通配符
Topic	同 Direct 类型, 但支持通配符。“*”匹配一个单字, “#”匹配零个或者多个单字, 单字之间是由“.”来分割的
Fanout	忽略 binding key 和 routing key, 消息会被传递到所有绑定的队列上

简单来说, Direct 是需要满足单一条件的路由, 在 Exchange 判断要将消息发送给哪个 Queue 时, 判断的依据只能是一个条件。Fanout 是指广播式的路由, 即将消息发送给所有的 Queue。而 Topic 是需要满足多个条件的路由, 转发消息时需要依据多个条件。

其他 Exchange 类型, 比如 Header、System 等在 OpenStack 中使用很少, 有兴趣的读者可以参考 AMQP 协议本身。

作为消息的存储和分发实体, Queue 会把消息缓存在内存或磁盘中, 并且按顺序把这些消息分发给一个或者多个消费者。

2. 基于 AMQP 实现 RPC

图 4-2 为基于 AMQP 实现远程过程调用 RPC 的过程。

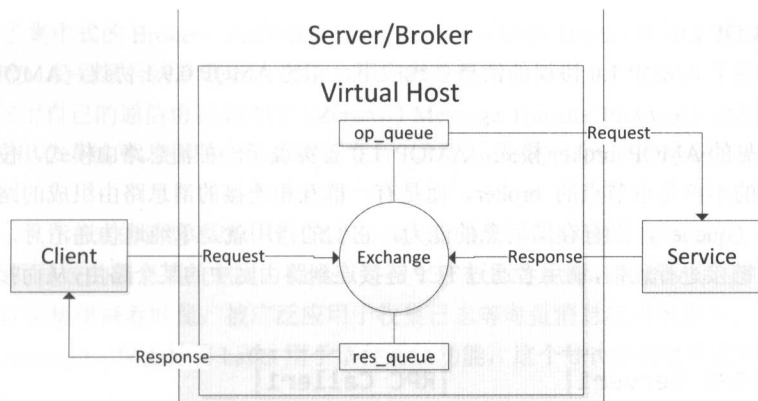


图 4-2 基于 AMQP 的 RPC 实现

- 客户端发送一个请求消息给 Exchange，指定 routing key 为 “op_queue”，同时指明一个消息队列名用来获取响应，图中为 “res_queue”。
- Exchange 把此消息转发到消息队列 op_queue。
- 消息队列 op_queue 把消息推送给服务端，服务端执行此 RPC 调用对应的任务。执行结束后，服务端把响应结果发送给消息队列，指明 routing key 为 “res_queue”。
- Exchange 把此消息转发到消息队列 res_queue。
- 客户端从消息队列 res_queue 中获取响应。

3. 常见的消息总线实现

如前所述，OpenStack 已经支持了一些常见的消息总线。oslo.messaging 通过实现不同的 driver 以支持不同的消息总线，如 RabbitMQ 和 ZeroMQ，也有一些 driver 旨在支持一类消息协议，如 AMQP 1.0 driver 支持所有采用 AMQP 1.0 协议的消息总线。

(1) RabbitMQ

RabbitMQ 是一个实现了 AMQP 的消息中间件服务。它包括 Server/Broker，支持多种协议的网关（HTTP、STOMP、MQTT 等），支持多种语言（Erlang、Java、.NET Framework 等）的客户端开发库，支持用户自定义插件开发的框架以及多种插件。

RabbitMQ 的 Server/Broker 使用 Erlang 语言编写，使用 Mozilla Public License (MPL) 许可证发行。详见 RabbitMQ 的主页 <http://www.rabbitmq.com/>。

oslo messaging 底层实现了两种不同的 driver 来支持 RabbitMQ，分别是 kombu 和 pika。它们的主要区别在于使用了不同的 Python library。pika library 对 RabbitMQ 的支持更加完备，所以在 Mitaka 版本加入了基于 pika library 的新 driver，以期能够使用更多的 RabbitMQ 特性，然而新 driver 尚需一定时间的检验，所以目前默认还是会采用 Kombu 来作为 RabbitMQ 的底层 library。

(2) AMQP 1.0

支持实现了 AMQP 1.0 协议的消息总线应用，相比 AMQP 0.9.1 协议，AMQP 1.0 更加灵活和复杂。

除了常见的 AMQP broker 模式，AMQP 1.0 还实现了一种消息路由模式，位于调用者和服务器之间的不再是单节点的 broker，而是有一群互相连接的消息路由组成的路由网。路由不具备队列（queue），没有存储消息的能力，它们的作用就是单纯地传递消息，路由节点之间使用 TCP 链接进行通信，调用者通过 TCP 链接连到路由网中的某个路由，从而接入路由网。

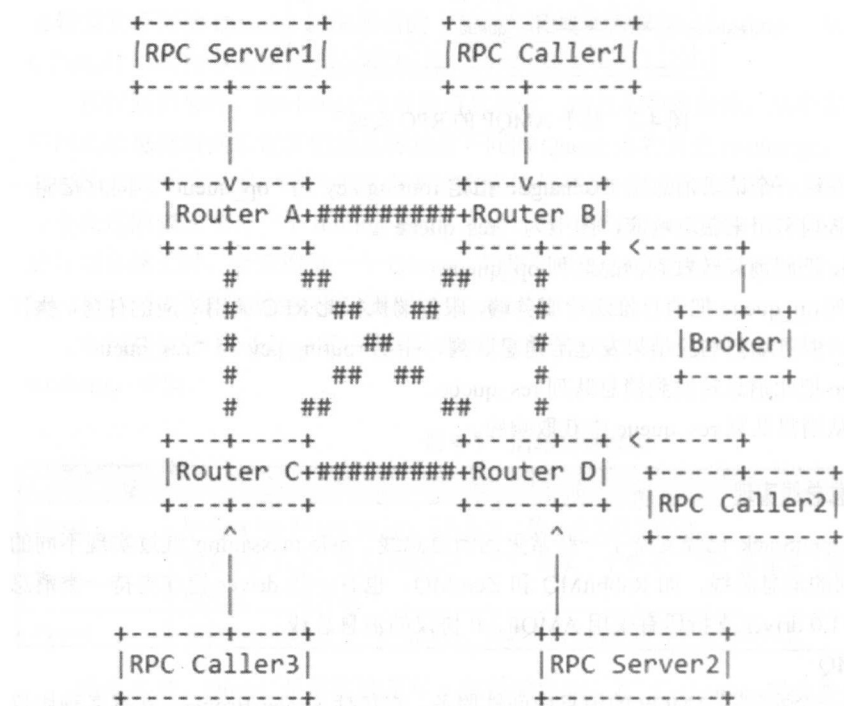


图 4-3 AMQP1.0 消息路由

如图 4-3 所示，当 RPC Caller1 远程调用 RPC Server2 上的某个方法时，消息会根据最短路径算法经过 RouterB 和 RouterD，最后到达 RPC Server2。

相比 broker，消息路由模式拥有更大的灵活性，可以通过向路由拓扑图中加入冗余节点来保证服务的高可用。

(3) ØMQ (ZeroMQ)

ZeroMQ 是一个开源的高性能异步消息库，和实现了 AMQP 的 RabbitMQ 和 Qpid 不同，ZeroMQ 系统可以在没有 Server/Broker 的情况下工作，消息发送者需要负责消息路由以找到正确的消息目的地，消息接收者需要负责消息的入队/出队等操作。

由于没有了集中式的 Broker，ZeroMQ 可以实现一般 AMQP Broker 所达不到的很低的延迟和较大的带宽，特别适合消息数量特别巨大的应用场景。

ZeroMQ 使用自己的通信协议 ZMTP (ZeroMQ Message Transfer Protocol) 来进行通信。ZeroMQ 的库使用 C++ 编写，使用 LGPL 许可证发行。详见 ZeroMQ 主页 <http://www.zeromq.org/>。

(4) Kafka

Kafka 是一个分布式的消息系统，使用 Scala 编写，最初由 LinkedIn 公司开发，现已成为 Apache 项目。与传统的消息系统相比，Kafka 是一个分布式的系统，有着较好的扩展能力，可以为发布和订阅提供高吞吐量，被广泛应用于收集日志等海量消息应用场景中。

目前 oslo.messaging 仅支持将 kafka 用于事件通知功能，这个 driver 尚处于实验阶段。

4.2 SQLAlchemy 和数据库

SQLAlchemy 是 Python 编程语言下的一款开源软件，使用 MIT 许可证发行。SQLAlchemy 提供了 SQL 工具包以及对象关系映射器 (Object Relational Mapper, ORM)，这样 SQLAlchemy 能让 Python 开发人员简单灵活地运用 SQL 操作后台数据库。

SQLAlchemy 主要分成两部分：SQLAlchemy Core (SQLAlchemy 核心) 和 SQLAlchemy ORM (SQLAlchemy 对象关系映射器)。SQLAlchemy Core 包括 SQL 语言表达式、数据引擎、连接池等，所有这一切的实现，都是为了连接不同类型的后台数据库、提交查询和更新 SQL 请求去后台执行、定义数据库数据类型和定义 Schema 等目的。SQLAlchemy ORM 提供数据映射模式，即把程序语言的对象数据映射成数据库中的关系数据，或把关系数据映射成对象数据。SQLAlchemy 架构如图 4-4 所示。

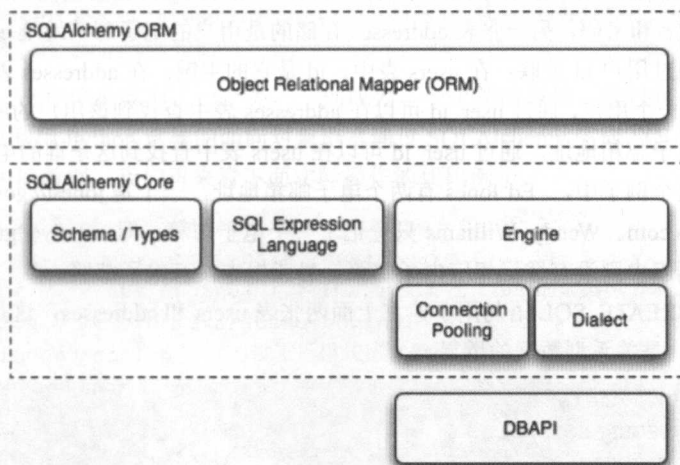


图 4-4 SQLAlchemy 架构

值得说明的是，如果程序用上了对象关系映射器，开发人员操作和理解数据就方便灵活了很多，但是，程序的性能会受到部分影响，毕竟，映射是需要额外开销的，因此，SQLAlchemy 中的对象关系映射是一个可选模块，开发人员在 Python 中完全可以不用任何对象模型也能直接用 SQLAlchemy 操作数据。

对象关系映射器在 Web 应用程序框架中也经常提到，因为它是快速开发栈中的关键组件。现代程序开发语言大多是面向对象的，而现今主流成熟的数据库系统基本上都是关系型数据库。所以，对象关系映射器主要解决的问题就是将面向对象型的程序操作映射成对数据库进行操作，而且把关系数据库的查询结果转成对象型数据便于程序访问。

这里是一个简单的例子，如果数据库中有两个表如表 4-2 与表 4-3 所示。

表 4-2 Table users

id	name	fullname	password
1	ed	Ed Jones	f8s7ccs
2	wendy	Wendy Williams	foobar
3	mary	Mary Contrary	xxg527
4	fred	Fred Flinstone	blah

表 4-3 Table addresses

id	user_id	email_address
1	1	jones@google.com
2	1	j25@yahoo.com
3	2	wendy@gmail.com

从关系模型来看，该关系型数据库中有两张表，一张表 users 对应的是用户信息，包括用户 id、名称、全名和密码，另一张表 addresses 存储的是用户的电子邮箱地址 email_address 信息。这两张表通过用户 id 关联：在 users 表中，id 是它的主键；在 addresses 表中，user_id 是它的外键。已知一个用户，通过 user_id 可以在 addresses 表中查找到该用户的所有电子邮箱地址；已知一个电子邮箱地址，通过 user_id 可以在 users 表中查找到这是谁的电子邮箱地址。

比如，在这个例子中，Ed Jones 有两个电子邮箱地址，一个是 jones@google.com，另一个是 j25@yahoo.com。Wendy Williams 只登记了一个电子邮箱，即 wendy@gmail.com，而其他用户在该数据库中则没有登记相应电子邮箱信息或根本没有电子邮箱。

下面两个 CREATE SQL 语句用于建立上面两张表 users 和 addresses，这是典型的关系数据库 SQL 语句，是关系型数据的世界。

```
CREATE TABLE users (  
    id INTEGER NOT NULL,  
    name VARCHAR,  
    fullname VARCHAR,  
    password VARCHAR,
```

```

        PRIMARY KEY (id)
    )
CREATE TABLE addresses (
    id INTEGER NOT NULL,
    email_address VARCHAR NOT NULL,
    user_id INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY (user_id) REFERENCES users (id)
)

```

从对象模型来看，则是另一个世界。同样是上面的例子，对象关系映射器可以把上面两张表映射成两个类 `class User` 和 `class Address`，它们的定义分别是：

```

>>> from sqlalchemy.ext.declarative import declarative_base
>>> from sqlalchemy import Column, Integer, String
>>> Base = declarative_base()
>>> class User(Base):
...     __tablename__ = 'users'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
>>> class Address(Base):
...     __tablename__ = 'addresses'
...     id = Column(Integer, primary_key=True)
...     email_address = Column(String, nullable=False)
...     user_id = Column(Integer, ForeignKey('users.id'))
...     user = relationship('User', backref=backref('addresses',
order_by=id))

```

粗略地说，一行记录成为了一个类，一列成为了一个类的属性。经过这样的模型转换和映射，我们可以利用 Python 这样的面向对象语言通过 SQLAlchemy 生成 SQL 语句，以便查询和更新数据库中的数据。我们再来看下面实际操作的代码例子。

```

>>> from sqlalchemy import create_engine
>>> from sqlalchemy.orm import sessionmaker
>>> engine = create_engine(...) #根据用户配置建立相应的数据库引擎
>>> Session = sessionmaker(bind=engine)
>>> session = Session() #通过工厂模式建立数据库 session
...
>>> for u, a in session.query(User, Address).\
...     filter(User.id==Address.user_id).\
...     filter(Address.email_address=='jones@google.com').\
...     all():
...     print u, a

```


这时候程序运行时，SQLAlchemy 会产生相应的 SELECT SQL 查询语句提交给后台数据库查询处理。后台生成的 SQL 查询语句如下所示。

```
SELECT users.id AS users_id,
       users.name AS users_name,
       users.fullname AS users_fullname,
       users.password AS users_password,
       addresses.id AS addresses_id,
       addresses.email_addresses AS addresses_email_address,
       addresses.user_id AS addresses_user_id
FROM users, addresses
WHERE users.id = addresses.user_id
      AND addresses.email_address = ?
('jones@google.com',)
```

打印出来的结果如下：

```
<User('ed', 'Ed Jones', 'f8s7ccs')> <Address('jones@google.com')>
```

细心的读者会发现，查询结果得到的其实是类 User 和类 Address 的某个实例，而要访问该实例的类属性，接下来可以直接使用 u.id、u.name、a.email_address 等面向对象程序化语句。这样，相当于完成了查询结果从关系型数据模型到对象型数据模型的映射。

至于使用 SQLAlchemy 完成其他操作，比如插入、更新和删除，也是类似的。下面是一个插入操作的例子。

```
>>> ed_user = User('yaed', 'Ed Jones', 'f8s7ccs')
>>> session.add(ed_user)
>>> session.flush()
```

此时 SQLAlchemy 会提交一个 INSERT SQL 语句给后台数据库，如下所示。而数据库 users 表中会多一条记录。

```
INSERT INTO users (name, fullname, password) VALUES (?, ?, ?)
('yaed', 'Ed Jones', 'edspassword')
```

SQLAlchemy 基本上支持绝大多数数据库 SQL 操作和特有属性。就拿上面的查询而言，由于表与表之间、类与类之间已经建立并定义了键与外键的关系，SQLAlchemy 可以直接利用 join() 函数来完成连接查询操作，而无须像上面的例子那样再次把查询条件一一列出。至于 join 操作是主动的还是被动的 (Lazy)，SQLAlchemy 也有一套参数可以供开发人员用程序进行选择、配置和控制。

另外，大多数数据库中的一些高级功能，比如事务处理 (transaction) 等，SQLAlchemy 也提供了相应的支持。也就是说，开发人员可以用 session 的 commit() 函数和 rollback() 函数告诉后台数据库，对刚才的数据库改动分别做提交和回退处理。

可以这么认为，SQLAlchemy 是一座架设在 Python 和各种各样后台数据库的桥梁，让开

发人员可以很容易且简便地用 Python 语句查询和更新数据库中的数据，而无须了解更多 SQL 语句的细节。更重要的是，如果后台数据库类型发生变化，假如数据从某类型的数据库管理系统迁移到另一种类型的数据库管理系统，开发人员的 Python 程序可以不用修改或做少量配置文件的修改仍可以正常运行。

在 OpenStack 中，有着大量的数据需要后台数据库保存和维护，比如虚拟机状态信息和各种监控数据，目前 OpenStack 可以提供 MySQL、Postgresql 等多种数据库做后台供选择，而操作它们基本上都用到了 SQLAlchemy 并进行了类封装。这些代码都保存在相应项目的 db 目录下，代码本身并不是太复杂，有兴趣的读者可以到相应的目录下查看。

目前 SQLAlchemy 已经发展到了 1.1.1 版本，不仅支持 Python 2.5 到最新的 Python 3.x 版本，而且支持 Jython 和 Pypy。就所支持的数据库而言，SQLAlchemy 已经支持 SQLite、Postgresql、MySQL、Oracle、MS-SQL、Firebird、Sybase 等多种数据库。

有关更加详细的 SQLAlchemy 说明文档，请参考 SQLAlchemy 官方网站 <http://www.sqlalchemy.org>。

4.3 RESTful API 和 WSGI

OpenStack 项目都是通过 RESTful API 向外提供服务，这使得 OpenStack 的接口在性能、可扩展性、可移植性、易用性等方面达到比较好的平衡。

1. 什么是 RESTful

RESTful 是目前流行的一种互联网软件架构。REST (Representational State Transfer, 表述性状态转移) 一词最早由 Roy Thomas Fielding 在他 2000 年的博士论文中提出，定义了他对互联网软件的架构原则，如果一个架构符合 REST 原则，就称它为 RESTful 架构。

RESTful 架构一个核心的概念是“资源”(Resource)。从 RESTful 的角度来看，网络里的任何东西都是资源，它可以是一段文本、一张图片、一首歌曲、一种服务等，每个资源都对应一个特定的 URI (统一资源定位符)，并用它进行标识，访问 URI 就可以获得这个资源。

资源可以有多种具体表现形式，也就是资源的“表述”(Representation)，比如一张图片既可以使用 JPEG 格式，也可以使用 PNG 格式。URI 只是代表了资源的实体，并不能代表它的表现形式。

互联网里，客户端和服务端之间的互动传递的就只是资源的表述，我们上网的过程，就是调用资源的 URI，获取它不同形式的过程。这个互动只能使用无状态协议 HTTP，也就是说，服务端必须保存所有的状态，客户端可以使用 HTTP 的几个基本操作，包括 GET (获取)、POST (创建)、PUT (更新) 与 DELETE (删除)，使服务端上的资源发生“状态转化”(State Transfer)，也就是所谓的“表述性状态转移”。

2. RESTful 路由

OpenStack 各个项目都提供了 RESTful 架构的 API 作为对外提供的接口，而 RESTful 架构的核心是资源与资源上的操作，这也就是说，OpenStack 定义了很多的资源，并实现了针对这些资源的各种操作函数。OpenStack 的 API 服务进程接收到客户端的 HTTP 请求时，一个所谓的“路由”模块会将请求的 URL 转换成相应的资源，并路由到合适的操作函数上。

OpenStack 中所使用的路由模块 Routes (<http://routes.readthedocs.org/>) 源自于对 Rails 路由系统的重新实现。Rails (Ruby on Rails) 是 Ruby 语言的 Web 开发框架，采用 MVC (Model-View-Controller) 模式，收到浏览器发出的 HTTP 请求后，Rails 路由系统会将这个请求指派到对应的 Controller。

```
# 新建一个 mapper 并创建路由
from routes import Mapper
map = Mapper()
map.connect(None, "/error/{action}/{id}", controller="error")
map.connect("home", "/", controller="main", action="index")

# URL '/error/myapp/4' 能够匹配上面的路由
result = map.match('/error/myapp/4')
# result == {'controller': 'error', 'action': 'myapp', 'id': '4'}
```

每个 Controller 都对应了一个 RESTful 资源，代表了对该资源的操作集合，其中包含很多个 Action (函数或者说操作)，比如 index、show、create、destroy 等，每个 Action 都对应着一个 HTTP 的请求和回应，比如执行“nova list”命令时，Nova 客户端 (novaclient) 将这个命令转换成 HTTP 请求发送给 Nova 的 API 服务进程，然后被路由到下面的“index”操作。

```
# nova/api/openstack/compute/servers.py

class ServersController(wsgi.Controller):

    @extensions.expected_errors((400, 403))
    def index(self, req):
        """返回虚拟机的列表给指定用户"""
        context = req.environ['nova.context']
        context.can(server_policies.SERVERS % 'index')
        try:
            servers = self._get_servers(req, is_detail=False)
        except exception.Invalid as err:
            raise exc.HTTPBadRequest(explanation=err.format_message())
        return servers
```

3. 什么是 WSGI

RESTful 只是设计风格而不是标准，Web 服务中通常使用基于 HTTP 的符合 RESTful 风

格的 API。而 WSGI (Web Server Gateway Interface, Web 服务器网关接口) 则是 Python 语言中所定义的 Web 服务器和 Web 应用程序或框架之间的通用接口标准。

从名称上看, WSGI 是一个网关, 作用就是在协议之间进行转换。换句话说, WSGI 就是一座桥梁, 桥梁的一端称为服务端或者网关端, 另一端称为应用端或者框架端。当处理一个 WSGI 请求时, 服务端为应用端提供上下文信息和一个回调函数, 应用端处理完请求后, 使用服务端所提供的回调函数返回相对应请求的响应。

作为一个桥梁, WSGI 将 Web 组件分成了 3 类: Web 服务器 (WSGI Server)、Web 中间件 (WSGI Middleware) 与 Web 应用程序 (WSGI Application)。WSGI Server 接收 HTTP 请求, 封装一系列环境变量, 按照 WSGI 接口标准调用注册的 WSGI Application, 最后将响应返回给客户端。

WSGI Application 是一个可被调用的 (Callable) Python 对象, 它接收两个参数, 通常为 `environ` 和 `start_response`。比如:

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    yield 'Hello World\n'
```

参数 `environ` 指向一个 Python 字典, 要求里面至少包含一些在 CGI (通用网关接口规范) 中定义的环境变量, 比如 `REQUEST_METHOD`、`SCRIPT_NAME`、`PATH_INFO`、`QUERY_STRING` 等。除此之外, `environ` 里面还至少要包含其他 7 个 WSGI 所定义的环境变量, 比如 `wsgi.version`、`wsgi.input`、`wsgi.url_scheme` 等。WSGI 应用可以从 `environ` 字典中获得相对应的请求及其执行上下文的所有信息。

参数 `start_response` 指向一个回调函数, 形如:

```
start_response(status, response_headers, exc_info=None)
```

`status` 参数是一个形如 “999 Message here” 的表示响应状态的字符串; `response_headers` 参数是一个包含了 (`header_name`, `header_value`) 元组的列表, 分别表示 HTTP 响应中的 HTTP 头及其内容; `exc_info` 一般在出现错误的时候使用, 用来让浏览器显示相关错误信息。

参数 `start_response` 所指向的这个回调函数需要返回另一个形如 `write(body_data)` 的可被调用对象。这个 `write` 对象是为了兼容现有的一些特殊框架设计的, 一般情况下不使用。

有请求到来时, WSGI Server 会准备好 `environ` 和 `start_response` 参数, 然后调用 WSGI Application 获得对应请求的响应。以下是一个 WSGI 服务端调用应用端的例子。

```
def call_application(app, environ):
    body = []
    status_headers = [None, None]
    # 定义 start_response 回调函数
    def start_response(status, headers):
        status_headers[:] = [status, headers]
        return body.append(status_headers)
```

```

# 调用 WSGI 应用端
app_iter = app(environ, start_response)
try:
    for item in app_iter:
        body.append(item)
finally:
    if hasattr(app_iter, 'close'):
        app_iter.close()
    return status_headers[0], status_headers[1], ''.join(body)
#准备 environ 环境变量, 假设 CGI 相关变量已经在操作系统的上下文中
environ = os.environ.items()
environ['wsgi.input'] = sys.stdin.buffer
environ['wsgi.errors'] = sys.stderr
environ['wsgi.version'] = (1, 0)
environ['wsgi.multithread'] = False
environ['wsgi.multiprocess'] = True
environ['wsgi.run_once'] = True
if environ.get('HTTPS', 'off') in ('on', '1'):
    environ['wsgi.url_scheme'] = 'https'
else:
    environ['wsgi.url_scheme'] = 'http'

status, headers, body = call_application(application, environ)

```

WSGI 中间件同时实现了服务端和应用端的 API, 因此可以在两端之间起协调作用。从服务端方面看, 中间件就是一个 WSGI 应用; 从应用端方面看, 中间件则是一个 WSGI 服务器。

WSGI 中间件可以将客户端的 HTTP 请求, 路由给不同的应用对象, 然后将应用处理后的结果返回给客户端。

我们也可以将 WSGI 中间件理解为服务端和应用端交互的一层包装, 经过不同中间件的包装, 便具有不同的功能, 比如 URL 路由分发, 再比如权限认证。这些不同中间件的组合便形成了 WSGI 的框架, 比如 Paste。

4. Paste

OpenStack 使用 Paste 的 Deploy 组件 (<http://pythonpaste.org/deploy/>) 来完成 WSGI 服务器和应用的构建, 每个项目源码的 etc 目录下都有一个 Paste 配置文件, 比如 Nova 中的 etc/nova/api-paste.ini, 部署时, 这些配置文件会被复制到系统/etc/<project>/目录下。Paste Deploy 的工作便是基于这些配置文件。

Paste 配置文件有其固有的格式, 这里以官网上的配置文件为例。

```

[composite:main]
use = egg:Paste#urlmap
/ = home

```



```

/blog = blog
/wiki = wiki
/cms = config:cms.ini

[app:home]
use = egg:Paste#static
document_root = %(here)s/htdocs

[filter-app:blog]
use = egg:Authentication#auth
next = blogapp
roles = admin
htpasswd = /home/me/users.htpasswd

[app:blogapp]
use = egg:BlogApp
database = sqlite:/home/me/blog.db

[app:wiki]
use = call:mywiki.main:application
database = sqlite:/home/me/wiki.db

```

Paste 配置文件分为多个 section，每个 section 以 type:name 的格式命名。

(1) type = composite

这个类型的 section 会把 URL 请求分发到对应的 Application，use 表明具体的分发方式，比如“egg:Paste#urlmap”表示使用 Paste 包中的 urlmap 模块，这个 section 里的其他形如“key = value”的行是使用 urlmap 进行分发时的参数。

(2) type = app

一个 app 就是一个具体的 WSGI Application，这个 app 对应的 Python 代码则由 use 来指定，共有两种指定的方法。

```

[app:myapp]
# 从另外一个 config.ini 文件中寻找 app
use = config:another_config_file.ini#app_name

[app:myotherapp]
# 从 Python EGG 中寻找
use = egg:MyApp

[app:mythirdapp]
# 直接调用另外一个模块中的 myapplication
use = call:my.project.myapplication

```



```
[app:mylastapp]
# 从另外一个 section 中
use = myotherapp
```

另外一种指定方法是明确指明对应的 Python 代码,这时必须给出代码所应该符合的格式,比如 `paste.app_factory`:

```
[app:myapp]
# myapp.modulename 将被加载,并从中获取 app_factory 对象
paste.app_factory = myapp.modulename:app_factory
```

Paste Deploy 定义了很多 factory, 这些 factory 只是为了便于使用针对 WSGI 标准的一些封装。比如最为普通的 `app_factory` 格式如下:

```
def composite_factory(loader, global_config, **local_conf):
    return wsgi_app
```

(3) type = filter-app

接收到一个请求后,会首先调用 `filter-app` 中的 `use` 所指定的 `app` 进行过滤,如果这个请求没有被过滤,就会被转发到 `next` 所指定的 `app` 进行下一步的处理。

(4) type = filter

与 `filter-app` 类型的区别只是没有 `next`。

(5) type = pipeline

`pipeline` 由一系列 `filter` 组成,这个 `filter` 链条的末尾是一个 `app`。`pipeline` 类型主要是对 `filter-app` 进行了简化,否则,如果有多个 `filter`,就需要写多个 `filter-app`,然后使用 `next` 进行连接。

```
[pipeline:main]
pipeline = filter1 egg:FilterEgg#filter2 filter3 app

[filter:filter1]
...
```

使用 Paste Deploy 的主要目的就是生成一个 WSGI Application,有了配置文件之后,只需要使用下面的调用方式:

```
from paste.deploy import loadapp
wsgi_app = loadapp('config:/path/to/config.ini')
```

对于 OpenStack,这里以 Nova 为例:

```
# nova/wsgi.py

from paste import deploy

class Loader(object):
```

```
"""从 Paste 配置文件加载 WSGI 应用"""
```

```
def load_app(self, name):
    try:
        LOG.debug("Loading app %(name)s from %(path)s",
                  {'name': name, 'path': self.config_path})
        return deploy.loadapp("config:%s" % self.config_path, name=name)
    except LookupError:
        LOG.exception(_LE("Couldn't lookup app: %s"), name)
        raise exception.PasteAppNotFound(name=name,
                                          path=self.config_path)
```

5. WebOb

除了 Routes 与 Paste Deploy 外, OpenStack 中另一个与 WSGI 密切相关的是 WebOb (<http://webob.org/>)。WebOb 通过对 WSGI 的请求与响应进行封装,来简化 WSGI 应用的编写。

WebOb 中两个最重要的对象,一是 webob.Request,对 WSGI 请求的 environ 参数进行封装;一是 webob.Response,包含了标准 WSGI 响应的所有要素。此外,还有一个 webob.exc 对象,针对 HTTP 错误代码进行封装。

除了这 3 个对象,WebOb 提供了一个修饰符(decorator),即 webob.dec.wsgify,以便可以不使用原始的 WSGI 参数和返回格式,而全部使用 WebOb 替代。

```
@wsgify
def myfunc(req):
    return webob.Response('hey there')
```

调用时可以有两种选择:

```
app_iter = myfunc(environ, start_response)
```

或

```
resp = myfunc(req)
```

第一种就是最原始和标准的 WSGI 格式,第二种则是 WebOb 封装过后的格式。

用户也可以使用参数对 wsgify 修饰符进行定制,比如使用 webob.Request 的子类,对真正的 Request 做一些判断或过滤,比如:

```
class MyRequest(webob.Request):
    @property
    def is_local(self):
        return self.remote_addr == '127.0.0.1'
@wsgify(RequestClass=MyRequest)
def myfunc(req):
    if req.is_local:
        return Response('hi!')
```

```

else:
    raise webob.exc.HTTPForbidden

```

以 Nova 为例:

```

# nova/wsgi.py

import webob.dec
import webob.exc

class Request(webob.Request):
    """继承 webob.Request"""
    def __init__(self, environ, *args, **kwargs):
        if CONF.wsgi.secure_proxy_ssl_header:
            scheme = environ.get(CONF.wsgi.secure_proxy_ssl_header)
            if scheme:
                environ['wsgi.url_scheme'] = scheme
        super(Request, self).__init__(environ, *args, **kwargs)

class Middleware(Application):
    """指定 wsgify 修饰符的参数"""
    @webob.dec.wsgify(RequestClass=Request)
    def __call__(self, req):
        response = self.process_request(req)
        if response:
            return response
        response = req.get_response(self.application)
        return self.process_response(response)

```

6. Pecan

随着 OpenStack 项目的发展, Paste 组合框架的 Restful API 代码的弊端也渐渐显现, 代码过于臃肿, 导致项目的可维护性变差。为了解决这个问题, 一些新项目选了 Pecan 框架来实现 Restful API。

Pecan 是一个轻量级的 WSGI 网络框架, 其设计并不想解决 Web 世界的所有问题, 而是主要集中在对象路由和 Restful 支持上, 并不提供对话 (session) 和数据库支持, 用户可以自由选择其他模块与之组合。

Pecan 的配置多位于 config.py 文件内, 以 ironic 项目为例:

```

# ironic/api/config.py

# 服务器相关配置
server = {

```

```

    'port': '6385',
    'host': '0.0.0.0'
}

# Pecan app 配置
app = {
    'root': 'ironic.api.controllers.root.RootController',
    'modules': ['ironic.api'],
    'static_root': '%(confdir)s/public',
    'debug': False,
    'acl_public_routes': [
        '/',
        '/v1',
        # IPA ramdisk methods
        '/v1/lookup',
        '/v1/heartbeat/[a-z0-9\~]+',
        # Old IPA ramdisk methods - will be removed in the Ocata release
        '/v1/drivers/[a-z0-9\~]*/vendor_passthru/lookup',
        '/v1/nodes/[a-z0-9\~]+/vendor_passthru/heartbeat',
    ],
}

# WSME 配置, 关闭 debug 模式
wsme = {
    'debug': False,
}

```

Pecan 的配置文件使用的也是 Python 语言, 每个配置项都是一个 Python 字典, 其中 server 指定了 WSGI 应用运行的主机和端口, app 指定了 WSGI app 有关的一些配置值。

modules 项是一个 Python 模块列表, Pecan 会在 modules 里寻找 WSGI 应用。Pecan 使用了对象路由的方式把一个 HTTP 请求映射到 Controller 的方法上。具体来说, 当用户访问某个 URL 的时候, Pecan 会将路径分割成许多部分, 从根控制器 (Root Controller) 开始沿着对象路径找到要执行的函数, root 项指定了根控制器的位置。以下是一个 Pecan 对象路由的例子。

```

from pecan import expose

class BooksController(object):
    @expose()
    def index(self):
        return "Welcome to book section."

    @expose()
    def bestsellers(self):
        return "We have 5 books in the top 10."

```

```

class CatalogController(object):
    @expose()
    def index(self):
        return "Welcome to the catalog."

    books = BooksController()

class RootController(object):
    @expose()
    def index(self):
        return "Welcome to store.example.com!"

    @expose()
    def hours(self):
        return "Open 24/7 on the web."

catalog = CatalogController()

```

当用户访问 `/catalog/books/bestsellers` 的时候，URL 会被分解成 `catalog`、`books`、`bestsellers` 三个部分，Pecan 会在 `Root Controller` 上寻找 `catalog`，继而在 `catalog` 对象上去查找 `books`，最终执行 `books` 对象上的 `bestsellers` 方法。

4.4 Eventlet

目前，OpenStack 中的绝大部分项目都采用所谓的协程（coroutine）模型。从操作系统的角度来看，一个 OpenStack 服务只会运行在一个进程中，但是在这个进程中，OpenStack 利用 Python 库 Eventlet 可以产生出许多个协程。这些协程之间只有在调用到了某些特殊的 Eventlet 库函数的时候（比如睡眠 `sleep`，I/O 调用等）才会发生切换。

与线程类似，协程也是一个执行序列，拥有自己独立的栈与局部变量，同时又与其他协程共享全局变量。协程与线程的主要区别是，多个线程可以同时运行，而同一时间内只能有一个协程在运行，无须考虑很多锁的问题，因此开发和调试也更简单方便。

使用线程时，线程的执行完全由操作系统控制，进程调度会决定什么时候哪个线程应该占用 CPU。而使用协程时，协程的执行顺序与时间完全由程序自己决定。如果某个工作比较耗费时间或需要等待某些资源，协程可以自己主动让出 CPU 进行休息，其他的协程工作一段时间后同样会主动把 CPU 让出，这样一来，我们可以控制各个任务的执行顺序，从而最大可能地利用 CPU 的性能。

协程的实现主要是在协程休息时把当前的寄存器保存起来，然后重新工作时再将其恢复，可以简单地理解为，在单个线程内部有多个栈去保存切换时的线程上下文，因此，协程可以理解为一个线程内的伪并发方式。

1. Eventlet

Eventlet(<http://eventlet.net>)是一个 Python 的网络库,它可以通过协程的方式来实现并发。Eventlet 将协程又称为 GreenThread (绿色线程)。所谓的并发,就是创建多个 GreenThread,并对其进行管理。

一个最简单的使用 Eventlet 的例子如下:

```
import eventlet

def my_func(param):
    .....
    return 0

gt = eventlet.spawn(my_func, work_to_process)
result = gt.wait()
```

eventlet.spawn 会新建一个 GreenThread 来运行 my_func 函数。由于 GreenThread 不会进行抢占式调度,所以此时这个新建的 GreenThread 只是被标识为可调度,并不会被立即调度执行。只有当主线程执行到 gt.wait()时,这个 GreenThread 才有机会被调度去执行 my_func 函数,进而开始自己的神奇之旅。

2. AsyncIO

由于 Eventlet 本身的一些局限性,比如不支持 Python3;只支持 CPython,不支持 PyPy 和 Jython 等。目前 OpenStack 社区正在考虑用 AsyncIO 来代替 eventlet。

AsyncIO 的设计标准定义在 PEP 3156 中,并且在 Python 3.4 中成为了标准内建模块,提供了一套用来写单线程并发代码的基础架构,其中包括协程、I/O 多路复用,以及信号量、队列、锁等一系列同步原语。AsyncIO 可以看做是许多第三方 Python 库的超集,包括 Twisted、Tornado、Gevent 和 Eventlet 等。

由于 OpenStack 的目标是支持从 Python 2.6 到 Python 3.5 的各个版本,而 AsyncIO 只在 Python 3.4 及其以后的版本中有支持,Enovance 公司开发了 trollius 库,把 AsyncIO 移植到了 Python 2.x 中, trollius 支持 Python 2.6 到 3.5 的所有版本。详情可以参见其官方网站 <https://bitbucket.org/enovance/trollius>。

4.5 OpenStack 通用库 Oslo

OpenStack 通用库 (Oslo) 包含了众多不需要重复发明的“轮子”。当开发者觉得现有的代码中有适合被其他 OpenStack 项目共用的部分时,可以向 oslo-specs 提交 blueprint 来提出创建新的 Oslo 项目。

4.5.1 Cliff

Cliff (Command Line Interface Formulation Framework) 可以用来帮助构建命令行程序。开发者利用 Cliff 框架可以构建诸如 svn、git 那样的支持多层命令的命令行程序。主程序只负责基本的命令行参数的解析, 然后调用各个子命令去执行不同的操作。利用 Python 动态代码载入的特性, Cliff 框架中的每个子命令可以和主程序分来地来实现、打包和分发。

Cliff 的代码库位于 <http://git.openstack.org/cgit/openstack/cliff>, 项目主页为 <https://launchpad.net/python-cliff>。

整个 Cliff 框架主要包括以下 4 种不同类型的对象。

- **cliff.app.App**: 主程序对象, 用来启动程序, 并且负责一些对所有子命令都通用的操作, 比如设置日志选项和输入/输出等。
- **cliff.commandmanager.CommandManager**: 主要用来载入每个子命令插件。默认是通过 Setuptools 的 entry points 来载入。
- **cliff.command.Command**: 用户可以实现 Command 的子类来实现不同的子命令, 这些子命令被注册在 Setuptools 的 entry points 中, 被 CommandManager 载入。每个子命令可以有自己的参数解析 (一般使用 argparse), 同时要实现 `take_action()` 方法完成具体的命令。
- **cliff.interactive.InteractiveApp**: 实现交互式命令行。一般使用框架提供的默认实现。

Cliff 源码中附带了一个示例 `demoapp`, 下面结合这个示例来了解 Cliff 的大致接口。

```
# main.py

import sys

from cliff.app import App
from cliff.commandmanager import CommandManager

class DemoApp(App):

    log = logging.getLogger(__name__)

    def __init__(self):
        super(DemoApp, self).__init__(
            description='cliff demo app',
            version='0.1',
            command_manager=CommandManager('cliff.demo'),
        )

    def initialize_app(self, argv):
        self.LOG.debug('initialize_app')
```

```

def prepare_to_run_command(self, cmd):
    self.LOG.debug('prepare_to_run_command %s',
                    cmd.__class__.__name__)

def clean_up(self, cmd, result, err):
    self.LOG.debug('clean_up %s', cmd.__class__.__name__)
    if err:
        self.LOG.debug('got an error: %s', err)

def main(argv=sys.argv[1:]):
    myapp = DemoApp()
    return myapp.run(argv)

if __name__ == '__main__':
    sys.exit(main(sys.argv[1:]))

```

上面是主程序的代码,新建一个 DemoAPP 对象实例,并且调用其 run 方法运行。DemoApp 是 cliff.app.App 的子类,它的初始化函数的原型定义如下:

```

class cliff.app.App(description, version, command_manager,
                    stdin=None, stdout=None, stderr=None,
                    interactive_app_factory=<class cliff.
                                         interactive.InteractiveApp>

```

其中,stdin/stdout/stderr 可以用来定义用户自己的标准输入/输出/错误,command_manager 必须指向一个 cliff.commandmanager.CommandManager 的对象实例。这个实例用来载入各个子命令插件。

cliff.commandmanager.CommandManager 类的初始化函数原型定义如下:

```

class cliff.commandmanager.CommandManager(namespace,
                                         convert_underscores=True)

```

其中,namespace 用来指定 Setuptools entry points 的命名空间,CommandManager 只会从这个命名空间中载入插件,convert_underscores 参数指明是否需要把 entry points 中的下画线转化为空格。

我们可以利用 cliff.app.App 类的方法 initialize_app()做一些初始化工作,这个函数会在主程序解析完用户的命令行参数后被调用,而且只会被调用到唯一一次。

prepare_to_run_command()方法可以被用来做一些针对某个具体子命令的初始化工作,它将在该子命令被执行之前调用。clean_up()方法会在具体某个子命令完成后被调用,用来进行一些清理工作。

具体某个子命令的实现通过继承 `cliff.command.Command` 来完成:

```
# simple.py

import logging

from cliff.command import Command

class Simple(Command):
    "A simple command that prints a message."

    log = logging.getLogger(__name__)

    def take_action(self, parsed_args):
        self.log.info('sending greeting')
        self.log.debug('debugging')
        self.app.stdout.write('hi!\n')
```

子命令的实际工作由 `take_action()` 完成。这个例子里, `simple` 子命令向标准输出打印一个字符串, 它的实现代码由 `cliff.commandmanager.CommandManager` 通过 `Setuptools entry points` 来载入。

```
# setup.py

from setuptools import setup, find_packages

setup(
    name='cliffdemo',
    version='0.1',
    .....
    install_requires=['cliff'],
    namespace_packages=[],
    packages=find_packages(),
    .....

    entry_points={
        'console_scripts': [
            'cliffdemo = cliffdemo.main:main'
        ],
        'cliff.demo': [
            'simple = cliffdemo.simple:Simple',
        ],
    },
)
```

在 Setuptools entry points 的命名空间 cliff.demo 中, 定义了命令 simple 所对应的插件实现是 Simple 类。Cliff 主程序解析用户的输入后, 会通过这里所定义的对对应关系调用不同的实现类。

simple 命令执行的结果如下:

```
$ cliffdemo simple
sending greeting
hi!

$ cliffdemo -v simple
prepare_to_run_command Simple
sending greeting
debugging
hi!
clean_up Simple
```

4.5.2 oslo.config

oslo.config 库用于解析命令行和配置文件中的配置选项, 代码库位于 <https://github.com/openstack/oslo.config>, 项目主页为 <https://launchpad.net/oslo.config>, 参考文档在 <http://docs.openstack.org/developer/oslo.config/>。

这里通过以下几个应用场景来介绍 oslo.config 的使用方法。

(1) 定义和注册配置选项

```
# file: service.py

from oslo.config import cfg
# cfg.CONF 是 oslo.config 中定义的一个全局对象实例

OPTS = [
    cfg.StrOpt('host',
               default=socket.gethostname(),
               help='Name of this node'),
    cfg.IntOpt('collector_workers',
               default=1,
               help='Number of workers for collector service.'),
]

# 注册配置选项
cfg.CONF.register_opts(OPTS)

# 将配置选项注册为命令行选项
CLI_OPTIONS = [
    cfg.StrOpt('os-tenant-id',
```



```

        deprecated_group="DEFAULT",
        default=os.environ.get('OS_TENANT_ID', ''),
        help='Tenant ID to use for OpenStack service access.',
        cfg.BoolOpt('insecure',
                    default=False,
                    help='Disables X.509 certificate validation when an '
                        'SSL connection to Identity Service is established.'),
    ]
    cfg.CONF.register_cli_opts(CLI_OPTIONS,
                              group="service_credentials")

```

配置选项有不同的类型，目前所支持的如表 4-4 所示。

表 4-4 oslo.config 支持的配置选项类型

类 名	说 明
oslo.config.cfg.StrOpt	字符串类型
oslo.config.cfg.BoolOpt	布尔型
oslo.config.cfg.IntOpt	整数类型
oslo.config.cfg.FloatOpt	浮点数类型
oslo.config.cfg.ListOpt	字符串列表类型
oslo.config.cfg.DictOpt	字典类型，字典中的值需要是字符串类型
oslo.config.cfg.MultiStrOpt	可以分多次配置的字符串列表
oslo.config.cfg.IPOpt	IP 地址类型
oslo.config.cfg.HostnameOpt	域名类型
oslo.config.cfg.URIOpt	URI 类型

定义后的配置选项，必须要注册才能使用。此外，配置选项还可以注册为命令行选项，之后，这些配置选项的值就可以从命令行读取，并覆盖从配置文件中读取的值。

注册配置选项时，可以把某些配置选项注册在一个特定的组下。这样可以帮助管理员更好地组织配置选项文件。如果没有指定，默认的组是“DEFAULT”。

在 1.3.0 版本的 oslo.config 中，增加了另一种新的定义配置选项的方式：

```

from oslo.config import cfg
from oslo.config import types

PortType = types.Integer(1, 65535)

common_opts = [
    cfg.Opt('bind_port',
            type=PortType(),
            default=9292,

```

```
help='Port number to listen on.')
```

```
1
```

相比于前面的方法,这种定义配置选项的方式能够更好地支持选项值的合法性检查,同时也能支持自定义选项类型。因此,建议新的项目用这种方式定义配置选项。但由于目前很多 OpenStack 项目还是在采用老方式。为了读者理解代码的方便,这里我们仍然采用老方式。

(2) 使用配置文件和命令行选项指定配置选项

为了正确使用 oslo.config, 应用程序一般需要在启动的时候初始化, 比如:

```
from oslo.config import cfg

conf(sys.argv[1:], project = 'xyz')
```

初始化后, 才能正常解析配置文件和命令行选项。最终用户可以用默认的命令行选项 “--config-file” 或者 “--config-dir” 来指定配置文件名或者位置。如果没有明确指定, 默认按下面的顺序寻找配置文件:

```
~/xyz/xyz.conf ~/xyz.conf /etc/xyz/xyz.conf /etc/xyz.conf
```

配置文件一般采用类似.ini 文件的格式, 其中每一个 Section 对应 oslo.config 中定义的一个配置选项组, Section [DEFAULT] 对应了默认组 “DEFAULT”。比如:

```
[DEFAULT]
host = 1.1.1.1
collector_workers = 3
[service_credentials]
insecure = True
```

用命令行指定配置选项值时, 如果是定义在某个选项组中的选项, 命令行选项名中需要包括该组名作为前缀:

```
--service_credentials-os-tenant-id ab23ef67
```

(3) 使用其他模块中已经注册过的配置选项

对于已经注册过的配置选项, 开发者可以直接访问:

```
from oslo.config import cfg
import service

hostname = cfg.CONF.host
tenant_id = cfg.CONF.service_credentials.os-tenant-id
```

这里导入 service 模块是因为选项 host 和 os-tenant-id 是在 service 模块中注册的。但是从编码风格来看, 上述代码比较古怪, 我们导入了 service 模块却从来没有直接使用它。所以, 我们也可以使用 import_opt 来申明在别的模块中定义的配置选项:

```
from oslo.config import cfg
```



```
cfg.CONF.import_opt('host', 'service')
hostname = cfg.CONF.host
```

4.5.3 oslo.db

oslo.db 是针对 SQLAlchemy 访问的抽象。代码库位于 <https://github.com/openstack/oslo.db>, 项目主页为 <https://bugs.launchpad.net/oslo>, 参考文档在 <http://docs.openstack.org/developer/oslo.db>。

这里我们通过几个不同的使用范例来了解 oslo.db 中主要接口的使用方法。

- 使用 SQLAlchemy 的 session 和 connection。

oslo.db 提供了 oslo_db.sqlalchemy.enginefacade 模块来获取 session 和 connection, 有两种方法来使用 enginefacade, 即函数装饰器 (decorator) 和上下文管理器 (context manager)。这两种调用方式都需要提供一个上下文对象。上下文对象可以是任何 Python 类。这样做的目的是提供一个统一规范的 session 使用模式, 避免调用者使用不当造成数据库事务 (transaction) 的滥用和嵌套。

```
from oslo.db.sqlalchemy import enginefacade

class MyContext(object):
    "User-defined context class."

def some_reader_api_function(context):
    with enginefacade.reader.using(context) as session:
        return session.query(SomeClass).all()

def some_writer_api_function(context, x, y):
    with enginefacade.writer.using(context) as session:
        session.add(SomeClass(x, y))

def run_some_database_calls():
    context = MyContext()

    results = some_reader_api_function(context)
    some_writer_api_function(context, 5, 10)
```

当使用装饰器模式的时候需要对 context 对象做特殊处理, 调用 transaction_context_provider 装饰 context 对象。

```
from oslo_db.sqlalchemy import enginefacade

@enginefacade.transaction_context_provider
class MyContext(object):
```

```

"User-defined context class."

@enginefacade.reader
def some_reader_api_function(context):
    return context.session.query(SomeClass).all()

@enginefacade.writer
def some_writer_api_function(context, x, y):
    context.session.add(SomeClass(x, y))

def run_some_database_calls():
    context = MyContext()

    results = some_reader_api_function(context)
    some_writer_api_function(context, 5, 10)

```

管理员可以通过配置文件来配置 `oslo.db` 的许多选项，比如：

```

[database]
connection = mysql://root:123456@localhost/ceilometer?charset=utf8

```

用户也可以在使用数据库之前调用 `oslo_db.sqlalchemy.enginefacade.configure` 方法来改变已有的配置。

常用的配置选项如表 4-5 所示（具体参见 `oslo/db/options.py`）。

表 4-5 常见的 `oslo.db` 配置选项

配置项 = 默认值	说 明
<code>backend = sqlalchemy</code>	（字符串类型）后台数据库标识
<code>connection = None</code>	（字符串类型） <code>sqlalchemy</code> 用此来连接数据库
<code>connection_debug = 0</code>	（整型） <code>sqlalchemy</code> 的 <code>debug</code> 等级，0 表示不输出任何调试信息，100 表示输出所有调试信息
<code>connection_trace = False</code>	（布尔型）是否把 Python 的调用栈信息加到 SQL 的注释中
<code>db_inc_retry_interval = True</code>	（布尔型）连接重试时，是否增加重试之间的时间间隔
<code>db_max_retries = 20</code>	（整型）连接重试的最多次数（-1 表示一直重试）
<code>db_max_retry_interval = 10</code>	（整型）连接重试时间间隔的最大值，单位为秒
<code>db_retry_interval = 1</code>	（整型）连接重试时间间隔，单位为秒
<code>idle_timeout = 3600</code>	（整型）连接被回收之前的空闲时间，单位为秒
<code>max_overflow = None</code>	（整型）如果设置了，这个参数会被直接传给 <code>sqlalchemy</code>
<code>max_pool_size = None</code>	（整型）在一个连接池中，最大可同时打开的连接数

续表

配置项 = 默认值	说 明
max_retries = 10	(整型) 打开连接时最大重试次数 (-1 表示一直重试)
retry_interval = 10	(整型) 打开连接时重试的间隔时间

- 使用 OpenStack 中通用的 SQLAlchemy model 类。

```
from oslo_db import models

class ProjectSomething(models.TimestampMixin, models.ModelBase):
    id = Column(Integer, primary_key=True)
    .....
```

oslo.db.models 模块目前只定义了两种 Mixin: TimestampMixin 和 SoftDeleteMixin。使用 TimestampMixin 时 SQLAlchemy model 中会多出两列 created_at 和 updated_at, 分别表示记录的创建时间和上一次修改时间。

SoftDeleteMixin 支持使用 soft delete 功能, 比如:

```
from oslo_db import models

class BarModel(models.SoftDeleteMixin, models.ModelBase):
    id = Column(Integer, primary_key=True)
    .....

count = model_query(BarModel).find(some_condition).soft_delete()
if count == 0:
    raise Exception("0 entries were soft deleted")
```

- 不同 DB 后端的支持。

```
from oslo_config import cfg
from oslo_db import api as db_api

# 定义不同 backend 所对应的实现, 如果配置选项 conf.database.backend
# 的值为 sqlalchemy, 就用 project.db.sqlalchemy.api 模块中的实现
_BACKEND_MAPPING = {'sqlalchemy': 'project.db.sqlalchemy.api'}

IMPL = db_api.DBAPI.from_config(cfg.CONF,
                                backend_mapping=_BACKEND_MAPPING)

def get_engine():
    return IMPL.get_engine()

def get_session():
```

```

        return IMPL.get_session()

# DB-API method
def do_something(somethind_id):
    return IMPL.do_something(somethind_id)

```

不同 backend 具体实现时，需要定义如下函数返回具体 DB API 的实现类：

```

def get_backend():
    return MyImplementationClass

```

4.5.4 oslo.i18n

oslo.i18n 是对 Python gettext 模块的封装，主要用于字符串的翻译和国际化。代码库位于 <https://github.com/openstack/oslo.i18n>，项目主页为 <https://launchpad.net/oslo.i18n>，参考文档在 <http://docs.openstack.org/developer/oslo.i18n/>。

使用 oslo.i18n 前，需要首先创建一个以下的集成模块：

```

# myapp/i18n.py

import oslo_i18n

_translators = oslo_i18n.TranslatorFactory(domain='myapp')

# 主要的翻译函数，类似 gettext 中的 “_” 函数
_ = _translators.primary

# 有上下文的翻译函数需要 oslo.i18n >=2.1.0
_C = _translators.contextual_form

# 复数形式的翻译函数需要 oslo.i18n >=2.1.0
_P = _translators.plural_form

# 不同的 log level 对应的翻译函数
# 注意，一般对于 debug level 的 log 信息，不建议翻译
_LI = _translators.log_info
_LW = _translators.log_warning
_LE = _translators.log_error
_LC = _translators.log_critical

```

之后，在程序中就可以比较容易使用：

```

from myapp.i18n import _, _LW, _LE

LOG.warn(_LW('warning message: %s'), var)
.....

```



```

try:
    .....
except Exception:
    LOG.exception(_LE('There was an error.'))
    .....
    raise ValueError(_('error: v1=%(v1)s v2=%(v2)s') % {'v1': v1, 'v2': v2})

```

4.5.5 oslo.messaging

oslo.messaging 库为 OpenStack 各个项目使用 RPC 和事件通知（Event Notification）提供了一套统一的接口。代码库位于 <https://github.com/openstack/oslo.messaging>，项目主页为 <https://launchpad.net/oslo.messaging>，参考文档在 <http://docs.openstack.org/developer/oslo.messaging>。

为了支持不同的 RPC 后端实现，oslo.messaging 对如下的对象进行了统一。

(1) Transport

Transport（传输层）主要实现 RPC 底层的通信（比如 Socket）以及事件循环、多线程等其他功能。用户可以通过 URL 来获得指向不同 transport 实现的句柄。URL 的格式如下：

```
transport://user:pass@host1:port[,hostN:portN]/virtual_host
```

目前支持的 Transport 有 rabbit、qpid 与 zmq，分别对应不同的后端消息总线。用户可以使用 oslo.messaging.get_transport 函数来获得 transport 对象实例的句柄。

(2) Target

Target 封装了指定某一个消息最终目的地的所有信息。表 4-6 所示为其所具有的属性。

表 4-6 Target 对象属性

参数=默认值	说 明
exchange = None	（字符串类型）topic 所属的范围，不指定的话默认使用配置文件中的 control_exchange 选项
topic = None	（字符串类型）一个 topic 可以用来标识服务器所暴露的一组接口（一个接口包含多个可被远程调用的方法）。允许多个服务器暴露同一组接口，消息会以轮转的方式发送给多个服务器中的某一个
namespace = None	（字符串类型）用来标识服务器所暴露的某个特定接口（多个可被远程调用的方法）
version = None	（字符串类型）服务器所暴露的接口支持 M.N 类型的版本号。次版本号（N）的增加表示新的接口向前兼容，主版本号（M）的增加表示新接口和旧接口不兼容。RPC 服务器可以实现多个不同的主版本号接口
server = None	（字符串类型）客户端可以指定此参数来要求消息的目的地是某个特定的服务器，而不是一组同属某个 topic 的服务器中的任意一台
fanout = None	（布尔型）当设置为真时，消息会被发送到同属某个 topic 的所有服务器上，而不是其中的一台

在不同的应用场景下，构造 Target 对象需要不同的参数：创建一个 RPC 服务器时，需要 topic 和 server 参数，exchange 参数可选；指定一个 endpoint 的 target 时，namespace 和 version 是可选的；客户端发送消息时，需要 topic 参数，其他可选。

(3) Server

一个 RPC 服务器可以暴露多个 endpoint，每个 endpoint 包含一组方法，这组方法可以被客户端通过某种 Transport 对象远程调用。创建 Server 对象时，需要指定 Transport、Target 和一组 endpoint。

(4) RPC Client

通过 RPC Client，可以远程调用 RPC Server 上的方法。远程调用时，需要提供一个字典对象来指明调用的上下文，调用方法的名字和传递给调用方法的参数（用字典表示）。

下面有 cast 和 call 两种远程调用方式。通过 cast 方式远程调用，请求发送后就直接返回了；通过 call 方式远程调用，需要等响应从服务器返回。

(5) Notifier

Notifier 用来通过某种 transport 发送通知消息。通知消息遵循如下的格式：

```
{'message_id': six.text_type(uuid.uuid4()), # 消息 id 号
 'publisher_id': 'compute.host1',          # 发送者 id
 'timestamp': timeutils.utcnow(),          # 时间戳
 'priority': 'WARN',                        # 通知优先级
 'event_type': 'compute.create_instance',  # 通知类型
 'payload': {'instance_id': 12, ... }      # 通知内容
}
```

用户可以在不同的优先级别上发送通知，这些优先级包括 sample、critical、error、warn、info、debug 和 audit 等。

(6) Notification Listener

Notification Listener 和 Server 类似，一个 Notification Listener 对象可以暴露多个 endpoint，每个 endpoint 包含一组方法。但是与 Server 对象中的 endpoint 不同的是，这里的 endpoint 中的方法对应通知消息的不同优先级。比如：

```
from oslo import messaging

class ErrorEndpoint(object):
    def error(self, ctxt, publisher_id, event_type, payload, metadata):
        do_something(payload)
        return messaging.NotificationResult.HANDLED
```

endpoint 中的方法如果返回 messaging.NotificationResult.HANDLED 或者 None，表示这个通知消息已经确认被处理；如果返回 messaging.NotificationResult.QUEUE，表示这个通知消息要重新进入消息队列。

下面是一个利用 oslo.messaging 来实现远程过程调用的示例。


```

# server.py 服务器端

from oslo.config import cfg
import oslo_messaging
import time

class ServerControlEndpoint(object):
    target = messaging.Target(namespace='control',
                              version='2.0')

    def __init__(self, server):
        self.server = server

    def stop(self, ctx):
        if self.server:
            self.server.stop()

class TestEndpoint(object):
    def test(self, ctx, arg):
        return arg

transport = oslo_messaging.get_transport(cfg.CONF)
target = oslo_messaging.Target(topic='test', server='server1')
endpoints = [
    ServerControlEndpoint(None),
    TestEndpoint(),
]
server = oslo_messaging.get_rpc_server(transport, target, endpoints,
                                       executor='blocking')

try:
    server.start()
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    print("Stopping server")

server.wait()

```

这个例子里，定义了两个不同的 endpoint：ServerControlEndpoint 与 TestEndpoint。这两个 endpoint 中的方法 stop() 和 test() 都可以被客户端远程调用。

创建 rpc server 对象之前，需要先创建 transport 和 target 对象，这里使用 get_transport() 函数来获得 transport 对象的句柄，get_transport() 的参数如表 4-7 所示。

表 4-7 get_transport()参数

参数=默认值	说 明
conf	(oslo.config.cfg.ConfigOpts 类型) oslo.config 配置项对象
url = None	(字符串或者 oslo.messaging.Transport 类型) transport URL。如果为空, 采用 conf 配置中的 transport_url 项所指定的值
allowed_remote_exmods = None	(列表类型) Python 模块的列表。客户端可用列表里的模块来 deserialize 异常
aliases = None	(字典类型) transport 别名和 transport 名称之间的对应关系

conf对象里,除了包含 transport_url 项外,还可以包含 control_exchange 项。control_exchange 用来指明 topic 所属的默认范围,默认值为“openstack”。用户可以使用 oslo.messaging.set_transport_defaults()函数来修改默认值。

此处构建的 Target 对象是用来建立 RPC server 的,所以需要指明 topic 和 server 参数。用户定义的 endpoint 对象也可以包含一个 target 属性,用来指明这个 endpoint 所支持的特定的 namespace 和 version。

这里使用 get_rpc_server()函数创建 server 对象,然后调用 server 对象的 start 方法开始接收远程调用。get_rpc_server()函数的参数如表 4-8 所示。

表 4-8 get_rpc_server()参数

参 数 = 默认值	说 明
transport	(Transport 类型) transport 对象
target	(Target 类型) target 对象, 用来指明监听的 exchange、topic 和 server
endpoints	(列表类型) 包含了 endpoints 对象实例的列表
executor = 'blocking'	(字符串类型) 用来指明消息接收和方法的方式。目前支持两种方式。 blocking: 在这种方式中, 用户调用 start 函数后, 在 start 函数中开始请求处理循环, 用户线程阻塞, 处理下一个请求。直到用户调用了 stop 函数后, 这个处理循环才会退出。消息的接收和分发处理都在调用 start 函数的线程中完成 eventlet: 在这种方式中, 会有一个协程 GreenThread 来处理消息的接收, 然后有其他不同的 GreenThread 来处理不同消息的分发处理。调用 start 的用户线程不会被阻塞
serializer = None	(Serializer 类型) 用来序列化/反序列化消息
access_policy = None	访问权限控制, 默认使用 LegacyRPCAccessPolicy

```
# client.py 客户端
```

```
from oslo.config import cfg
import oslo_messaging
```

```

transport = oslo_messaging.get_transport(cfg.CONF)
target = oslo_messaging.Target(topic='test')
client = oslo_messaging.RPCClient(transport, target)
ret = client.call(ctxt = {},
                  method = 'test',
                  arg = 'myarg')

ctxt = client.prepare(namespace='control', version='2.0')
ctxt.cast({}, 'stop')

```

这里 `target` 对象构造时，必须要有的参数只有 `topic`，创建 `RPCClient` 对象时，可以接收的参数如表 4-9 所示。

表 4-9 `RPCClient` 对象参数

参数=默认值	说 明
<code>transport</code>	(<code>Transport</code> 类型) <code>transport</code> 对象
<code>target</code>	(<code>Target</code> 类型) 该 <code>client</code> 对象的默认 <code>target</code> 对象
<code>timeout = None</code>	(整数或者浮点数类型) 客户端调用 <code>call</code> 方法时超时时间(单位为秒)
<code>version_cap = None</code>	(字符串类型) 最大所支持的版本号。当版本号超过时，会抛出 <code>RPCVersionCapError</code> 异常
<code>serializer = None</code>	(<code>Serializer</code> 类型) 用来序列化/反序列化消息
<code>retry = None</code>	(整数类型) 连接重试次数 None 或者 -1: 一直重试 0: 不重试 >0: 重试次数

远程调用时，需要传入调用上下文、调用方法的名字和传给调用方法的参数。

`Target` 对象的属性在 `RPCClient` 对象构造以后，还可以通过 `prepare()` 方法修改。用户可以修改的属性包括 `exchange`、`topic`、`namespace`、`version`、`server`、`fanout`、`timeout`、`version_cap` 和 `retry`。修改后的 `target` 属性只在 `prepare()` 方法返回的对象中有效。

下面我们再来看一个利用 `oslo.messaging` 实现通知消息处理的例子：

```

# notification_listener.py 消息通知处理

from oslo.config import cfg
import oslo_messaging

class NotificationEndpoint(object):
    filter_rule = oslo_messaging.NotificationFilter(
        publisher_id='^compute.*')

```

```

def warn(self, ctxt, publisher_id, event_type, payload, metadata):
    do_something(payload)

class ErrorEndpoint(object):
    filter_rule = oslo_messaging.NotificationFilter(
        event_type='^instance\..*\..start$',
        context={'ctxt_key': 'regex'})

    def error(self, ctxt, publisher_id, event_type, payload, metadata):
        do_something(payload)

transport = oslo_messaging.get_transport(cfg.CONF)
targets = [
    oslo_messaging.Target(topic='notifications')
    oslo_messaging.Target(topic='notifications_bis')
]
endpoints = [
    NotificationEndpoint(),
    ErrorEndpoint(),
]
pool = "listener-workers"
listener = oslo_messaging.get_notification_listener(transport, targets,
                                                    endpoints, pool=pool)

listener.start()
listener.wait()

```

通知消息处理的 endpoint 对象和远程过程调用的 ednpoint 对象不同，对象定义的方法需要和通知消息的优先级一一对应。我们可以为每个 endpoint 指定所对应的 target 对象。

最后调用 get_notification_listener()函数构造 notification listener 对象，get_notification_listener()函数的参数如表 4-10 所示。

表 4-10 get_notification_listener()参数

参 数 = 默认值	说 明
transport	(Transport 类型) transport 对象
target	(列表类型) target 对象的列表，用来指明 endpoints 列表中的每一个 endpoint 所侦听处理的 exchange 和 topic
endpoints	(列表类型) 包含 endpoints 对象实例的列表
executor = 'blocking'	(字符串类型) 用来指明消息接收和方法的方式：目前支持 3 种方式：blocking、eventlet 和 threading

续表

参 数 = 默认值	说 明
	<p>blocking: 在这种方式中, 用户调用 <code>start</code> 函数后, 在 <code>start</code> 函数中开始请求处理循环。用户线程阻塞, 处理下一个请求, 直到用户调用了 <code>stop</code> 函数后, 这个处理循环才会退出。消息的接收和分发处理都在调用 <code>start</code> 函数的线程中完成</p> <p>eventlet: 在这种方式中, 会有一个协程 <code>greenthread</code> 来处理消息的接收, 然后有其他不同的 <code>greenthread</code> 来处理不同消息的分发处理。调用 <code>start</code> 的用户线程不会被阻塞</p>
<code>serializer = None</code>	(<code>Serializer</code> 类型) 用来序列化/反序列化消息
<code>allow_requeue = False</code>	(布尔类型) 如果为真, 表示支持 <code>NotificationResult.REQUEUE</code>
<code>pool = None</code>	(字符串类型) 当多个 <code>listener</code> 订阅相同的 <code>target</code> 对象时, 可以通过设置 <code>pool</code> 改变消息传递的模式, 部分 <code>driver</code> 不支持设置 <code>pool</code>

相对应的发送消息通知的代码如下:

```
# notifier_send.py

from oslo.config import cfg
import oslo_messaging

transport = oslo_messaging.get_transport(cfg.CONF)
notifier = messaging.Notifier(transport,
                              driver='messaging',
                              topic='notifications')
notifier2 = notifier.prepare(publisher_id='compute')
notifier2.error(ctxt={},
                event_type='my_type',
                payload = {'content': 'error occurred'})
```

发送通知消息时, 首先要构造 `Notifier` 对象, 此时可能需要指定的参数如表 4-11 所示。

表 4-11 Notifer 对象参数

参 数 = 默认值	说 明
<code>transport</code>	(<code>Transport</code> 类型) <code>transport</code> 对象
<code>topics = None</code>	(字符串列表) 发送消息的 <code>topic</code> 列表, 如果没有指定, 会使用配置文件中的 <code>oslo_messaging_notifications</code> 段下 <code>topics</code> 的值
<code>publish_id = None</code>	(字符串类型) 发送者 <code>id</code>
<code>driver = None</code>	(字符串类型) 后台驱动。一般采用 <code>messaging</code> 。如果没有指定, 会使用配置文件中的 <code>notification_driver</code> 的值

续表

参 数 = 默认值	说 明
topic = None	(字符串类型) 发送消息的 topic。如果没有指定, 会使用配置文件中的 notification_topics 的值
serializer = None	(Serializer 类型) 用来序列化/反序列化消息
retry = None	(整数类型) 连接重试次数 None 或者 -1: 一直重试 0: 不重试 >0: 重试次数

初始化 Notifier 对象的操作比较复杂, 所以可以用 prepare()方法修改已创建的 Notifier 对象, prepare()方法返回的是新的 Notifier 对象的实例。它的参数如表 4-12 所示。

表 4-12 prepare ()参数

参 数 = 默认值	说 明
publish_id = None	(字符串类型) 发送者 id
retry = None	(整数类型) 连接重试次数 None 或者 -1: 一直重试 0: 不重试 >0: 重试次数

最后可以调用 Notifier 对象的不同方法 (error, critical, warn, 等等) 发送不同优先级的消息通知。

4.5.6 stevedore

利用 Python 语言的特性, 运行时动态载入代码变得更加容易。很多 Python 应用程序利用这样的特性在运行时发现和载入所谓的“插件”(plugin), 使得自己更易于扩展。Python 库 stevedore 就是在 Setuptools 的 entry points 基础上, 构造了一层抽象层, 使开发者可以更容易地在运行时发现和载入插件。

stevedore 的代码库在 <https://github.com/openstack/stevedore>, 项目主页在 <https://launchpad.net/python-stevedore>, 参考文档在 <http://stevedore.readthedocs.org/>。

entry points 的每一个命名空间里, 可以包含多个 entry point 项。stevedore 要求每一项都符合如下格式:

```
name = module:importable
```

左边是插件的名字, 右边是它的具体实现, 中间用等号分隔开。插件的具体实现用“模块:可导入的对象”的形式来指定, 以 Ceilometer 为例:


```

ceilometer.compute.virt =
    libvirt = ceilometer.compute.virt.libvirt.inspector:LibvirtInspector
    hyperv = ceilometer.compute.virt.hyperv.inspector:HyperVInspector
    vsphere = ceilometer.compute.virt.vmware.inspector:VsphereInspector
    xenapi = ceilometer.compute.virt.xenapi.inspector:XenapiInspector

ceilometer.hardware.inspectors =
    snmp = ceilometer.hardware.inspector.snmp:SNMPInspector

```

示例中显示了两个不同的 entry points 的命名空间，“ceilometer.compute.virt”和“ceilometer.hardware.inspectors”，分别注册有 4 个和 1 个插件。每个插件都符合“名字 = 模块:可导入对象”的格式，在“ceilometer.compute.virt”命名空间里的 libvirt 插件，它的具体可载入的实现是 ceilometer.compute.virt.libvirt.inspector 模块中的 LibvirtInspector 类。

根据每个插件在 entry point 中名字和具体实现的数量之间的对应关系不同，stevedore 提供了多种不同的类来帮助开发者发现和载入插件，如表 4-13 所示。

表 4-13 插件名字和具体实现的对应关系

插件名字: 具体实现	建议选用 stevedore 中的类
1: 1	stevedore.driver.DriverManager
1: n	stevedore.hook.HookManager
n: m	stevedore.extension.ExtensionManager

使用 stevedore 来帮助程序动态载入插件的过程主要分为 3 个部分：插件的实现、插件的注册和插件的载入。下面以 Ceilometer 里动态载入 compute agent 上 inspector 驱动为例来分别进行介绍。

1. 插件的实现

Ceilometer 的 inspector 驱动，为从不同类型 hypervisor 中获取相关数据提供统一的接口以供 compute agent 调用。下面是它的基类：

```

# ceilometer/compute/virt/inspector.py

class Inspector(object):

    def inspect_cpus(self, instance_name):
        """Inspect the CPU statistics for an instance.

        :param instance_name: the name of the target instance
        :return: the number of CPUs and cumulative CPU time
        """
        raise NotImplementedError()

```

.....

ceilometer/compute/virt/libvirt/inspector.py、ceilometer/compute/virt/hyperv/inspector.py、ceilometer/compute/virt/vmware/inspector.py 和 ceilometer/compute/virt/xenapi/inspector.py 分别为 kvm、hyperv、vsphere 和 xenapi 4 种不同 hypervisor 的具体实现，比如：

```
# ceilometer/compute/virt/libvirt/inspector.py

from ceilometer.compute.virt import inspector as virt_inspector

class LibvirtInspector(virt_inspector.Inspector):

    per_type_uris = dict(uml='uml:///system',
                        xen='xen:/// ',
                        lxc='lxc:///')

    def __init__(self):
        self.uri = self._get_uri()
        self.connection = None

    .....

    def inspect_cpus(self, instance_name):
        domain = self._lookup_by_name(instance_name)
        dom_info = domain.info()
        return virt_inspector.CPUStats(number=dom_info[3],
time=dom_info[4])

    .....
```

2. 插件的注册

上述的插件需要在 Setuptools 的相关文件中注册后，才能被 stevedore 库所认识。

```
# setup.cfg

ceilometer.compute.virt =
    libvirt = ceilometer.compute.virt.libvirt.inspector:LibvirtInspector
    hyperv = ceilometer.compute.virt.hyperv.inspector:HyperVInspector
    vsphere = ceilometer.compute.virt.vmware.inspector:VsphereInspector
    xenapi = ceilometer.compute.virt.xenapi.inspector:XenapiInspector
```

这四个插件注册在命名空间“ceilometer.compute.virt”下，分别叫做 libvirt、hyperv、vsphere 和 xenapi。

3. 插件的载入

```
# ceilometer/compute/virt/inspector.py

from oslo.config import cfg
from stevedore import driver

def get_hypervisor_inspector():
    try:
        namespace = 'ceilometer.compute.virt'
        mgr = driver.DriverManager(namespace,
                                   cfg.CONF.hypervisor_inspector,
                                   invoke_on_load=True)

        return mgr.driver
    except ImportError as e:
        LOG.error(_("Unable to load the hypervisor inspector: %s") % (e))
    return Inspector()
```

Ceilometer 的 compute agent 通过调用函数 `get_hypervisor_inspector` 来载入具体的某一个插件。此处由于插件和具体实现之间是一对一的关系，所以选用了 `stevedore` 的 `DriverManager` 类。这个类实例化时可接收的参数如表 4-14 所示：

表 4-14 DriverManager 类实例化时可接收的参数

参 数 = 默认值	说 明
<code>namespace</code>	(字符串类型) 命名空间
<code>name</code>	(字符串类型) 插件名
<code>invoke_on_load = False</code>	(布尔类型) 是否调用 <code>entrypoint</code> 所返回的插件对象。(在这个例子中，由于 <code>entrypoint</code> 所指向的对象是类，相当于是否实例化类对象)
<code>invoke_arg s=()</code>	(元组类型) 调用插件对象时所需的位置参数 (positional argument)
<code>invoke_kw d s= {}</code>	(字典类型) 调用插件对象时所需的命名参数 (named argument)
<code>on_load_failure_callback= None</code>	(函数类型) 载入某个 <code>entrypoint</code> 失败时的回调函数，参数为 (manager, <code>entrypoint</code> , exception)
<code>verify_requirements = False</code>	(布尔类型) 是否用 <code>setuptools</code> 来确保此插件的依赖关系都能满足

注意这里我们使用的命名空间 “`ceilometer.compute.virt`” 需要和 `Setuptools` 中注册的命名空间名称一致。具体需要载入的插件名称从配置选项 `hypervisor_inspector` 中读入。

4.5.7 TaskFlow

通过 `TaskFlow` 库，可以更容易地控制任务 (Task) 的执行。代码库在 <https://github.com/openstack/taskflow>，项目主页是 <http://launchpad.net/taskflow>，文档在 <http://docs.openstack.org/developer/taskflow/>。

1. task、flow 和 engine

我们利用下面的示例来了解 TaskFlow 中几个最基本的概念——task、flow 和 engine。

```
import taskflow.engines
from taskflow.patterns import linear_flow as lf
from taskflow import task

class CallJim(task.Task):
    def execute(self, jim_number, *args, **kwargs):
        print("Calling jim %s." % jim_number)

    def revert(self, jim_number, *args, **kwargs):
        print("Calling %s and apologizing." % jim_number)

class CallJoe(task.Task):
    def execute(self, joe_number, *args, **kwargs):
        print("Calling joe %s." % joe_number)

    def revert(self, joe_number, *args, **kwargs):
        print("Calling %s and apologizing." % joe_number)

class CallSuzzie(task.Task):
    def execute(self, suzzie_number, *args, **kwargs):
        raise IOError("Suzzie not home right now.")

flow = lf.Flow('simple-linear').add(
    CallJim(),
    CallJoe(),
    CallSuzzie()
)
try:
    taskflow.engines.run(flow,
                           engine_conf = {'engine': 'serial'},
                           store=dict(joe_number=444,
                                       jim_number=555,
                                       suzzie_number=666))
except Exception as e:
    print("Flow failed: %s" % e)
```

这个示例首先定义了 3 个 task: CallJim、CallJoe 和 CallSuzzie。在 TaskFlow 库中, task 是拥有执行 (execute) 和回滚 (revert) 功能的最小单位 (TaskFlow 中的最小单位是 atom, 其他所有类包括 Task 都是 Atom 类的子类)。在 Task 类中, 允许开发者定义自己的 execute 函数

和 revert 函数，分别用来执行 task 和回退 task 到之前一次的执行结果。

然后新建一个线性流 flow，并在其中顺序加入上述 3 个 task 对象。TaskFlow 中的流 flow 用来关联各个 task，并且规范这些 task 之间的执行和回滚顺序。Taskflow 中所支持的流类型如表 4-15 所示。

表 4-15 Taskflow 支持的 flow 类型

流 类 型	说 明
linear_flow.Flow	线性流，流中的 task/flow 按加入顺序执行，按加入顺序的倒序回滚
unordered_flow.Flow	无顺序流，流中的 task/flow 的执行和回滚可以按任意顺序
graph_flow.Flow	图流，流中的 task/flow 按照显式指定的依赖关系，或者通过其间 provides 和 requires 属性之间的隐含依赖关系，来执行或回滚

这个示例中，由于采用的是线性流，所以这个流中 task 执行的顺序为 CallJim→CallJoe→CallSuzzie，回滚的顺序是其倒序。由此，它的输出结果如下：

```
Calling jim 555.  
Calling joe 444.  
Calling 444 and apologizing.  
Calling 555 and apologizing.  
Flow failed: Suzzie not home right now.
```

流中不仅可以加入任务，还可以嵌套加入其他的流。此外，流还可以通过 retry 来控制当错误发生时，如何进行重试。TaskFlow 自带的 retry 类型如表 4-16 所示。

表 4-16 Taskflow 支持的 retry 类型

Retry 类型	说 明
AlwaysRevert	错误发生时，回滚子流
AlwaysRevertAll	错误发生时，回滚所有的流
Times	错误发生时，重试子流
ForEach	每次错误发生时，为子流中的 atom 提供一个新的值，然后重试，直到成功或者此 retry 中定义的值用光为止
ParameterizedForEach	类似 ForEach，但是是从后台存储中获取重试的值

比如下面的示例中，构造了一个线性流 f1，它按顺序执行任务 t1、子线性流 f2 和任务 t4。子流 f2 按序执行任务 t2 和 t3。

子流 f2 中定义了 ForEach 类型的 retry “r1”。当任务 t2 或者 t3 失败时，子流 f2 首先会回滚。然后“r1”会指导子流 f2 使用值“a”来重新运行。如果再次失败，子流 f2 回滚后会再次使用值“b”运行；仍然失败后回滚使用值“c”运行。如果值“c”也运行失败，由于“r1”中能提供的值已经被用完，子流 f2 回滚后不会重行运行。

```
flow = linear_flow.Flow('f1').add(  
    EchoTask('t1'),
```

```
linear_flow.Flow('f2', retry=retry.ForEach(values=['a', 'b', 'c'],
                                              name='r1', provides='value')).add(
    EchoTask('t2'),
    EchoTask('t3', requires='value')),
EchoTask('t4'))
```

TaskFlow 库中的 engine 用来载入一个 flow，然后驱动该 flow 中的 task/flow 运行。我们可以通过 engine_conf 来指明不同的 engine 类型，如表 4-17 所示。

表 4-17 Taskflow 支持的 engine 类型

engine 类型	说 明
'serial'	所有的 task 都在调用 engine.run 的那个线程中运行
'parallel'	task 可能会被调度到不同的线程中并发运行
'worker-based'	task 会被调度到不同的 worker 中运行。一个 worker 是一个单独的专门用来运行某些特定 task 的进程，这个 worker 进程可以在远程机器上，利用 AMQP 来通信

2. task 和 flow 的输入/输出

我们利用另外一个示例来了解 task 和 flow 的输入和输出。

```
import taskflow.engines
from taskflow.patterns import graph_flow as gf
from taskflow.patterns import linear_flow as lf
from taskflow import task

class Adder(task.Task):
    def execute(self, x, y):
        return x + y

flow = gf.Flow('root').add(
    lf.Flow('nested_linear').add(
        # 从后台存储中读取名为 y3 和 y4 的参数值，并以参数 x,y 传递给其 execute 方法
        # x2 = y3+y4 = 12
        Adder("add2", provides='x2', rebind=['y3', 'y4']),
        # x1 = y1+y2 = 4
        Adder("add1", provides='x1', rebind=['y1', 'y2'])
    ),
    # x5 = x1+x3 = 20
    Adder("add5", provides='x5', rebind=['x1', 'x3']),
    # x3 = x1+x2 = 16
    Adder("add3", provides='x3', rebind=['x1', 'x2']),
    # x4 = x2+y5 = 21
```



```

    Adder("add4", provides='x4', rebind=['x2', 'y5']),
    # x6 = x5+x4 = 41
    Adder("add6", provides='x6', rebind=['x5', 'x4']),
    # x7 = x6+x6 = 82
    Adder("add7", provides='x7', rebind=['x6', 'x6']))

# 为流 root 提供所需要的输入参数
store = {
    "y1": 1,
    "y2": 3,
    "y3": 5,
    "y4": 7,
    "y5": 9,
}

result = taskflow.engines.run(
    flow, engine_conf='serial', store=store)
print("Single threaded engine result %s" % result)

result = taskflow.engines.run(
    flow, engine_conf='parallel', store=store)
print("Multi threaded engine result %s" % result)

```

上面的例子中，定义了一个 Task 对象 Adder，作用是完成一个加法。接下去生成了一个图类型的流 root，其中的 task 都通过 provides 和 rebind 来指明它们的输出和输入。

在 engine 运行时，通过 store 参数为流 root 提供所需要的输入参数，engine 会把 store 中的值保存在后台存储中；在执行各个 task 的过程中，各个 task 的输入都从后台存储中获取，输出都保存在后台存储中。这个程序的输出结果如下：

```

Single threaded engine result {'y2': 3, 'x6': 41, 'y4': 7, 'y1': 1, 'x2':
12, 'x3': 16, 'y3': 5, 'x1': 4, 'y5': 9, 'x7': 82, 'x4': 21, 'x5': 20}
Multi threaded engine result {'y2': 3, 'x6': 41, 'y4': 7, 'y1': 1, 'x2':
12, 'x3': 16, 'y3': 5, 'x1': 4, 'y5': 9, 'x7': 82, 'x4': 21, 'x5': 20}

```

如上所述，TaskFlow 中的 Task 和 Retry 都是 Atom 的子类。对于任一种 Atom 对象，都可以通过 requires 属性来了解它所要求的输入参数，和通过 provides 属性来了解它能够提供的输出结果的名字。requires 和 provides 的类型都是包含参数名称的集合（set）。

Task 对象的 requires 可以由其 execute 方法获得。比如上述示例中的 Adder 对象，由于 execute 方法的参数是 execute(self, x, y)，所以它的 requires 为：

```

>>> Adder().requires
set(['y', 'x'])

```

注意，execute 方法中的可选参数和 *args 和 **kwargs 并不会出现在 requires 中：

```
>>> class MyTask(task.Task):
...     def execute(self, spam, eggs=()):
...         return spam + eggs
...
>>> MyTask().requires
set(['spam'])
>>>
>>> class UniTask(task.Task):
...     def execute(self, *args, **kwargs):
...         pass
...
>>> UniTask().requires
set([])
```

此外,也可以在创建 task 时明确指定它的输入参数要求,这些参数在调用 execute 方法时可以通过 kwargs 获得:

```
>>> class Dog(task.Task):
...     def execute(self, food, **kwargs):
...         pass
>>> dog = Dog(requires=('water', 'grass'))
>>> sorted(dog.requires)
['food', 'grass', 'water']
```

在有些情况下,传递给某个 task 的输入参数名称和其所需要的参数名不同,这个时候可以通过 rebind 来处理。

```
class SpawnVMTask(task.Task):
    def execute(self, vm_name, vm_image_id, **kwargs):
        pass

# engine 执行下面这个 task 时,会从后台存储中获取名为'name'的参数值
# 然后把它当做 vm_name 参数传递给 task 的 execute() 方法
SpawnVMTask(rebind={'vm_name': 'name'})

# engine 执行下面这个 task 时,会从后台存储中获取名为'name', 'image_id'
# 和 'admin_key_name' 的参数值,把 name 和 image_id 的值分别当做 vm_name
# 和 vm_image_id 参数,把 admin_key_name 当做 kwargs 参数中的某一项传递
# 给 task 的 execute 方法
SpawnVMTask(rebind=('name', 'image_id', 'admin_key_name'))
```

task 的输出结果一般是指其 execute 方法的返回值。但是由于 Python 的返回值是没有名字的,所以需要通过 Task 对象的 provides 属性指明返回值以什么名称存入后台存储中。根据 execute 的返回值类型不同,provides 可以有不同的方式指定。

- 如果 execute 方法返回的是一个单一的值:

```

class TheAnswerReturningTask(task.Task):
    def execute(self):
        return 42

# 指明此 task 的返回值以名称 the_answer 保存在后台存储中
TheAnswerReturningTask(provides='the_answer')

# 此 task 执行完毕后
>>> storage.fetch('the_answer')
24

```

- 如果 execute 方法返回元组 tuple:

```

class BitsAndPiecesTask(task.Task):
    def execute(self):
        return 'BITS', 'PIECES'

# 指明此 task 的返回值分别以名称 bits 和 pieces 保存在后台存储中
BitsAndPiecesTask(provides=('bits', 'pieces'))

# 此 task 执行完毕后
>>> storage.fetch('bits')
'BITS'
>>> storage.fetch('pieces')
'PIECES'

```

- 如果 execute 方法返回一个字典:

```

class BitsAndPiecesTask(task.Task):
    def execute(self):
        return {
            'bits': 'BITS',
            'pieces': 'PIECES'
        }

# provides 是 set 类型表示返回的是字典类型
BitsAndPiecesTask(provides=set(['bits', 'pieces']))

# 此 task 执行完毕后
>>> storage.fetch('bits')
'BITS'
>>> storage.fetch('pieces')
'PIECES'

```

4.5.8 cookiecutter

用户可以利用在 <https://github.com/openstack-dev/cookiecutter> 的模板，新建一个符合惯例的 OpenStack 项目。

```
# 安装 cookiecutter
$ sudo pip install cookiecutter

# 利用 cookiecutter 模板新建 openstack 项目
$ cookiecutter https://git.openstack.org/openstack-dev/cookiecutter.git
Cloning into 'cookiecutter'...
module_name (default is "replace with the name of the python module")? abc
repo_group (default is "openstack")? stackforge
repo_name (default is "replace with the name for the git repo")? abc
launchpad_project (default is "replace with the name of the project on
launchpad")? abc
project_short_description (default is "OpenStack Boilerplate contains all
the boilerplate you need to create an OpenStack package.")? "test project for
OpenStack"

# 初始化 git 代码库
$ cd abc
$ git init
Initialized empty Git repository in /tmp/abc/.git/
$ git add .
$ git commit -a

$ ls
abc          LICENSE          setup.cfg
babel.cfg    MANIFEST.in      setup.py
CONTRIBUTING.rst openstack-common.conf test-requirements.txt
doc          README.rst       tox.ini
HACKING.rst  requirements.txt
```

我们可以看到利用 cookiecutter 模板建立起来的项目中，顶层目录下包含如表 4-18 所示的文件和目录。

表 4-18 基于 cookiecutter 模板的项目结构

文 件	说 明
abc	代码目录
babel.cfg	babel 配置文件。babel 是一个用来帮助代码国家化的工具
CONTRIBUTING.rst	开发者文件
doc	文档目录

续表

文 件	说 明
HACKING.rst	编码规范文件
LICENSE	项目许可证信息
MANIFEST.in	MANIFEST 模板文件
openstack-common.conf	项目所用到的 oslo-incubator 库里的模块
README.rst	项目说明文件
requirements.txt	项目所依赖的第三方 python 库
setup.cfg	setuptools 配置文件
setup.py	setuptools 主文件
test-requirements.txt	项目测试时需要依赖的第三方 python 库
tox.ini	项目测试的 tox 环境配置文件

4.5.9 oslo.policy

policy 用于控制用户的权限，能够执行什么样的操作。OpenStack 的每个项目都有一个 `/etc/<project>/policy.json` 文件，通过配置这个文件来实现对用户的权限管理。

将 policy 操作的公共部分提取出来，就形成了 oslo.policy 库，它会负责 policy 的验证和 rules 的管理。一条 Rule 有两种形式，既可以是列表的列表，也可以是 policy 自定义的形式。Policy 模块中有专门的两个方法对两种格式的 Rules 解析。

Rules 的两种格式如下：

```
[["role:admin"],["project_id:%(project_id)s", "role:projectadmin"]]
role:admin or (project_id:%(project_id)s and role:projectadmin)
```

使用第二种格式，policy 规则支持 or、and 和 not 等逻辑的组合，而且还可以是带有“http”的 url 形式的 Rules。

policy 的验证，其实就是对字典 key 和 value 的判断，如果匹配成功，则通过 policy，否则失败。

各个工程的 API 通过 policy 来检测用户身份群权限的规则，例如有些 API 只有管理员权限可以执行，有些普通用户可以执行，在代码中的体现就是判断 context 的 project_id 和 user_id 是不是合法类型的。以下是 Nova API 的一个示例。

```
# nova/policy.py

def authorize(context, action, target, do_raise=True, exc=None):
    """验证用户行为的权限"""
    init()
```

```

credentials = context.to_policy_values()
if not exc:
    exc = exception.PolicyNotAuthorized
try:
    result = _ENFORCER.authorize(action, target, credentials,
                                do_raise=do_raise, exc=exc, action=action)
except policy.PolicyNotRegistered:
    with excutils.save_and_reraise_exception():
        LOG.exception(_LE('Policy not registered'))
except Exception:
    with excutils.save_and_reraise_exception():
        LOG.debug('Policy check for %(action)s failed with credentials '
                  '%(credentials)s',
                  {'action': action, 'credentials': credentials})
return result

```

相应/etc/nova/policy.json 文件内容如下:

```

"context_is_admin": "role:admin",
"admin_or_owner": "is_admin:True or project_id:%(project_id)s",
"admin_api": "is_admin:True"
.....

```

上面的例子可以看到, nova pause 的 rule 是 “is_admin:True or project_id:%(project_id)s”, 需要 policy 验证是不是 admin 用户或者 project_id 是不是匹配。

4.5.10 oslo.rootwrap

oslo.rootwrap 可以让其他 OpenStack 服务以 root 身份执行 shell 命令。一般来说, OpenStack 的服务都是以非特权用户的身份运行的, 但是当它们需要以 root 身份运行某些 shell 命令时, 就需要利用到 oslo.rootwrap 的功能。

oslo.rootwrap 首先会从配置文件所定义的 Filter 文件目录中读入所有 Filter 的定义, 然后检查要运行的 shell 命令是否和 Filter 中的定义相匹配, 匹配则运行, 不匹配就不运行。

1. 构造 rootwrap shell 脚本

使用 rootwrap 需要在一个单独的 Python 进程中以 root 身份调用 Python 函数 oslo.rootwrap.cmd.main()。我们可以通过 Setuptools 中的 console script 来构造这样一个 shell 脚本, 以 nova 为例:

```

# setup.cfg

console_scripts =
    nova-all = nova.cmd.all:main
.....

```



```
nova-rootwrap = oslo.rootwrap.cmd:main
```

可以看到构造一个名为 nova-rootwrap 的 shell 脚本时，会调用 oslo.rootwrap.cmd.main() 函数。运行 “python setup.py install” 之后，nova-rootwrap 脚本就会被生成。

2. 调用 rootwrap shell 脚本

rootwrap 的 shell 脚本需要以 sudo 方式调用，比如：

```
sudo nova-rootwrap /etc/nova/rootwrap.conf COMMAND_LINE
```

其中的/etc/nova/rootwrap.conf 是 oslo.rootwrap 的配置文件名，COMMAND_LINE 是希望以 root 用户身份运行的 shell 命令。

由于 rootwrap shell 脚本需要以 sudo 方式运行，所以还需要配置 sudoers 文件：

```
nova ALL = (root) NOPASSWD: /usr/bin/nova-rootwrap /etc/nova/rootwrap.conf *
```

这里我们假设 nova 服务一般以用户 nova 的身份运行，相关的 rootwrap shell 脚本是 /usr/bin/nova-rootwrap。

3. rootwrap 配置文件

rootwrap 配置文件是以 INI 的文件格式存放的。表 4-19 所示为相关的配置选项。

表 4-19 rootwrap 配置选项

选项 = 默认值	说 明
filters_path	包含 Filter 定义文件的目录，用逗号分隔，比如 filters_path=/etc/nova/rootwrap.d,/usr/share/nova/rootwrap
exec_dir=\$PATH	shell 可执行命令的搜索目录，用逗号分隔，比如 exec_dirs=/sbin,/usr/sbin,/bin,/usr/bin。默认使用系统环境变量 PATH 中的值
use_syslog=False	是否使用 syslog
syslog_log_facility=syslog	syslog 的 facility level，可选的其他选项有 auth、authpriv、syslog、user0 和 user1 等
syslog_log_level=ERROR	需要记录的 syslog 等级

4. 定义 Filter

Filter 定义文件一般以 .filters 后缀结尾，放在配置选项 filters_path 所指定的目录中。这些定义文件以 ini 格式存放，Filter 的定义放在 [Filters] 节中。定义的格式如下：

```
Filter 名: Filter 类, [Filter 类参数 1, Filter 类参数 2, ...]
```

rootwrap 目前所支持的 Filter 类型如表 4-20 所示。

表 4-20 rootwrap 所支持的 Filter 类型

Filter class	说 明
CommandFilter	<p>只检查运行的 shell 命令。类参数如下：</p> <ul style="list-style-type: none"> ● 可运行的 shell 命令 ● 用什么用户身份运行此命令 <p>比如下面的定义允许以 root 用户身份运行 kpartx 命令：</p> <p>kpartx: CommandFilter, kpartx, root</p>
RegExpFilter	<p>首先检查运行的 shell 命令，然后用正则表达式检查所有的命令参数。类参数如下：</p> <ul style="list-style-type: none"> ● 可运行的 shell 命令 ● 用什么用户身份运行此命令 ● 用以匹配第一个命令行参数的正则表达式 ● 用以匹配第二个命令行参数的正则表达式 <p>比如下面的定义允许以 root 用户运行 /usr/sbin/tunctl，运行时只允许有 3 个参数，并且第一个和第二个分别是 -b 和 -t：</p> <p>tunctl: /usr/sbin/tunctl, root, tunctl, -b, -t, .*</p>
PathFilter	<p>检查命令行参数中的目录是否合法。类参数如下：</p> <ul style="list-style-type: none"> ● 可运行的 shell 命令 ● 用什么用户身份运行此命令 ● 第一个命令行参数 ● 第二个命令行参数 <p>此处命令行参数可以有 3 种不同类型的参数定义</p> <p>pass: 允许任何命令行参数</p> <p>以 “/” 开头的字符串：命令行参数里的目录是在此目录下</p> <p>其他字符串：只允许此字符串为命令行参数</p> <p>比如下面的定义允许对 /var/lib/images 下的任何文件运行 chown nova 命令：</p> <p>chown: PathFilter, /bin/chown, root, nova, /var/lib/images</p>
EnvFilter	<p>允许设置额外的环境变量。类参数如下：</p> <ul style="list-style-type: none"> ● env ● 用什么用户身份运行此命令 ● (多个) 允许设置的环境变量名，用 “=” 结尾 ● 可运行的 shell 命令 <p>比如下面的定义允许以 root 用户运行诸如 “CONFIG_FILE=foo NETWORK_ID=bar dnsmasq” 的命令：</p> <p>dnsmasq: EnvFilter, env, root, CONFIG_FILE=, NETWORK_ID=, dnsmasq</p>

续表

Filter class	说 明
ReadFileFilter	<p>允许使用 cat 来读取文件。类参数如下：</p> <ul style="list-style-type: none"> ● 允许以 root 用户的身份读取的文件 <p>比如下面的定义允许以 root 用户运行 “cat /foo/bar”：</p> <p>read_initiator: ReadFileFilter, /foo/bar</p>
KillFilter	<p>允许对特定进程发送特定信号。类参数如下：</p> <ul style="list-style-type: none"> ● 用什么用户身份运行 kill 命令 ● 只向执行此命令的进程发送信号 ● (多个) 允许发送的信号 <p>比如下面的定义允许向 /usr/sbin/dnsmasq 进程发送信号-9 或者-HUP：</p> <p>kill_dnsmasq: KillFilter, root, /usr/sbin/dnsmasq, -9, -HUP</p>
IpFilter	<p>允许运行 ip 命令（除了 ip netns exec 之外）。类参数如下：</p> <ul style="list-style-type: none"> ● ip ● 用什么用户身份运行 ip 命令 <p>ip: IpFilter, ip, root</p>
IpNetnsExecFilter	<p>允许运行 ip netns exec <namespace> <command>命令，但是其中的<command>必须要通过其他的 Filter 的检查。类参数如下：</p> <ul style="list-style-type: none"> ● ip ● 用什么用户身份运行 ip 命令 <p>ip: IpNetnsExecFilter, ip, root</p>
ChainingRegExpFilter	<p>ChainingRegExpFilter 首先使用 RegExpFilter 类的方式检查在此类参数定义的前面几个命令行参数，剩下的命令行参数由其他 filter 定义检查。类参数如下：</p> <ul style="list-style-type: none"> ● 可运行的 shell 命令 ● 用什么用户身份运行此命令 ● (多个) 命令行参数 <p>比如下面的定义允许以 root 用户运行 /usr/bin/nice，但是第一个参数必须是-n，第二个参数必须是整数，接下去的参数由其他的 filter 检查：</p> <p>nice: ChainingRegExpFilter, /usr/bin/nice, root, nice, -n, -?\d+</p>

4.5.11 oslo.test

oslo.test 库提供单元测试的基础框架。代码库在 <https://github.com/openstack/oslotest>，文档在 <http://docs.openstack.org/developer/oslotest/>。

oslo.test 基于 testtools 库定义了 oslotest.base.BaseTestCase 类，可以作为其他 OpenStack 项目单元测试类的基类，比如：

```
from oslotest import base

class MyTestCases(base.BaseTestCase):
    def setUp(self):
```

```

super(MyTestCases, self).setUp()
//my setup things
.....

```

使用 `BaseTestCase` 作为基类时，单元测试中创建的所有临时文件都会被存放在一个单独的目录中，此时系统环境变量 `HOME` 也会被设置成一个临时的目录。

`BaseTestCase` 类提供了方法 `create_tempfiles()` 来创建临时文件：

```

create_tempfiles(files, ext='.conf')

```

参数： `files` (元组的列表) - 包含了类似 (文件名, 文件内容) 的元组的列表
`ext` (字符串) - 新建文件扩展名
返回：所有新建的文件名列表

此外，使用 `BaseTestCase`，还可以通过设置如表 4-21 所示的环境变量来控制单元测试中的一些功能。

表 4-21 控制单元测试功能的环境变量

环境变量	说 明
<code>OS_TEST_TIMEOUT</code>	如果设置的环境变量为整数，可以控制单元测试用例的最长可运行时间。超过了此时间限制的测试用例会被认为失败
<code>OS_STDOUT_CAPTURE</code>	如果此环境变量为真，则用一个假的流对象代替系统标准输出 <code>stdout</code>
<code>OS_STDERR_CAPTURE</code>	如果此环境变量为真，则用一个假的流对象代替系统标准输出错误 <code>stderr</code>
<code>OS_DEBUG</code>	如果此环境变量为真，则 <code>logging level</code> 会被设置成 <code>DEBUG</code> 等级
<code>OS_LOG_CAPTURE</code>	如果此环境变量为真，会用一个假的 <code>logging</code> 对象代替 <code>python</code> 的 <code>logging</code>

`oslo.test` 库除了提供上面的基类之外，还提供了两个通用的 `fixture`，即 `oslotest.mockpatch` 和 `oslotest.moxstubout`，供其他 `OpenStack` 项目开发单元测试用例。一般建议使用 `oslotest.mockpatch`。比如：

```

from oslotest import mockpatch

def setUp(self):
    super(TestSNMPInspector, self).setUp()
    self.inspector = snmp.SNMPInspector()
    # 用 faux_getCmd 函数替换 self.inspector._cmdGen.getCmd 函数
    self.useFixture(mockpatch.PatchObject(
        self.inspector._cmdGen, 'getCmd', new=faux_getCmd))
    .....

# 调用 keystoneclient.v2_0.client.Client 返回的类对象实例会被替换
# mock.Mock 类对象实例
self.useFixture(mockpatch.Patch(
    'keystoneclient.v2_0.client.Client',
    return_value=mock.Mock()))

```

OsloTest 库还提供一个 debug 脚本用来支持在测试中使用 pdb。

(1) 在代码中设置断点

```
import pdb; pdb.set_trace()
```

(2) 在 OpenStack 项目的 tox.ini 配置文件中加入如下行

```
[testenv:debug]
commands = oslo_debug_helper.sh {posargs}
```

(3) 运行类似下面的命令触发断点，进入 python debugger

```
$ tox -e debug
$ tox -e debug test_collector
$ tox -e debug test_collector.TestCollector
$ tox -e debug test_collector.TestCollector.test_only_rpc
```

4.5.12 oslo.versionedobjects

我们知道，在项目的不断迭代和升级之中，数据库结构和 API 接口的改动不可避免，如果没有一个版本控制的概念在里面，新旧不同的模块之间就很容易交互出现问题。oslo.versionedobjects 库提供了一种通用的自带版本的对象模型，自带序列化功能，可以很容易地和 oslo.messaging 结合进行远程调用。

使用 oslo.versionedobjects 构建独立于外部 API 和数据库的数据模型，可以很好地保证数据库结构升级后的兼容性。代码库在 <http://git.openstack.org/cgit/openstack/oslo.versionedobjects>，文档在 <http://docs.openstack.org/developer/oslo.versionedobjects>。

oslo.versionedobjects 代码多位于项目的 objects 目录下，通常包括 base.py、fields.py 以及不同资源的抽象对象文件。下面以 Nova 为例：

```
# nova/objects/base.py

from oslo_versionedobjects import base as ovoo_base
.....

from nova.objects import fields as obj_fields

class NovaObject(ovoo_base.VersionedObject):
    # 用于序列化的名空间
    OBJ_SERIAL_NAMESPACE = 'nova_object'
    OBJ_PROJECT_NAMESPACE = 'nova'

class NovaObjectDictCompat(ovoo_base.VersionedObjectDictCompat):
    def __iter__(self):
        for name in self.obj_fields:
            if (self.obj_attr_is_set(name) or
```



```

        name in self.obj_extra_fields):
            yield name

    def keys(self):
        return list(self)

class NovaTimestampObject(object):
    fields = {
        'created_at': obj_fields.DateTimeField(nullable=True),
        'updated_at': obj_fields.DateTimeField(nullable=True),
    }

class NovaPersistentObject(object):
    fields = {
        'created_at': obj_fields.DateTimeField(nullable=True),
        'updated_at': obj_fields.DateTimeField(nullable=True),
        'deleted_at': obj_fields.DateTimeField(nullable=True),
        'deleted': obj_fields.BooleanField(default=False),
    }

```

这里 nova 创建了自己的版本对象 NovaObject，继承 VersionedObject，所有需要版本化的 Nova 资源对象，例如 instance、flavor、security group 等，都可以继承 NovaObject 来定义自己的对象。

NovaObjectDictCompat, NovaTimestampObject 与 NovaPersistentObject 抽象了某些特有的需求，可以在构建 NovaObject 子类时作为 Mixin 使用。NovaObjectDictCompat 为对象提供了 dict 的一些接口，可以像操作 dict 一样去使用类实例。通过设置 versionedobject 的 fields 属性，NovaTimestampObject 包含了数据库中的时间戳字段，NovaPersistentObject 则包括持久化到数据库里的必备字段，子类继承这两个 Mixin 可以省去定义这些字段的工作。

versionedobject 既然是数据模型的一种抽象，理所当然也会包括对象里的各种字段，这些字段的抽象一般位于 objects 目录下的 fields.py 文件。

```

# nova/objects/fields.py

from oslo_versionedobjects import fields

BooleanField = fields.BooleanField
UnspecifiedDefault = fields.UnspecifiedDefault
IntegerField = fields.IntegerField
UUIDField = fields.UUIDField
FloatField = fields.FloatField
StringField = fields.StringField

```



```

.....
Enum = fields.Enum

class BaseNovaEnum(Enum):
    def __init__(self, **kwargs):
        super(BaseNovaEnum, self).__init__(valid_values=self.__class__.ALL)

class Architecture(BaseNovaEnum):
    ALL = arch.ALL

    def coerce(self, obj, attr, value):
        try:
            value = arch.canonicalize(value)
        except exception.InvalidArchitectureName:
            msg = _("Architecture name '%s' is not valid") % value
            raise ValueError(msg)
        return super(Architecture, self).coerce(obj, attr, value)
.....

```

`fields.py` 文件里定义了所有需要的字段类型，除了使用 `oslo.versionedobjects` 自带的常用类型，也通过继承常用类型实现了许多 Nova 需要的复杂字段，通过这些字段的组合我们就拥有了定义各种资源对象模型的能力。

```

# nova/objects/instance.py

from nova.objects import fields

# register 装饰器对定义的 versionedobjects 进行注册
@base.NovaObjectRegistry.register
class Instance(base.NovaPersistentObject, base.NovaObject,
               base.NovaObjectDictCompat):
    # 不同版本的注释
    # Version 2.0: Initial version
    # Version 2.1: Added services
    # Version 2.2: Added keypairs
    # Version 2.3: Added device_metadata
    VERSION = '2.3'

    fields = {
        'id': fields.IntegerField(),
        'user_id': fields.StringField(nullable=True),
        'project_id': fields.StringField(nullable=True),
    }

```

```

        'image_ref': fields.StringField(nullable=True),
        'kernel_id': fields.StringField(nullable=True),
        'ramdisk_id': fields.StringField(nullable=True),
        'hostname': fields.StringField(nullable=True),
        .....
    }

    def obj_make_compatible(self, primitive, target_version):
        super(Instance, self).obj_make_compatible(primitive, target_version)
        target_version = versionutils.convert_version_to_tuple(target_version)
        if target_version < (2, 3) and 'device_metadata' in primitive:
            del primitive['device_metadata']
        if target_version < (2, 2) and 'keypairs' in primitive:
            del primitive['keypairs']
        if target_version < (2, 1) and 'services' in primitive:
            del primitive['services']

    # remotable_classmethod 装饰的函数具备了远程调用的能力
    @base.remotable_classmethod
    def get_by_uuid(cls, context, uuid, expected_attrs=None, use_slave=False):
        if expected_attrs is None:
            expected_attrs = ['info_cache', 'security_groups']
        columns_to_join = _expected_cols(expected_attrs)
        db_inst = cls._db_instance_get_by_uuid(context, uuid,
                                                columns_to_join,
                                                use_slave=use_slave)
        return cls._from_db_object(context, cls(), db_inst,
                                    expected_attrs)

```

通过继承 NovaObject 和 Mixin 类, 创建了 Instance 的数据模型。Instance 类实现了一些接口供调用者查询 Instance 实例, 如 get_by_uuid 函数, 通过给定的 instance uuid 返回数据库中的 Instance 信息。remotable_classmethod 是 oslo.versionedobjects 提供的 RPC 装饰器, 在 Nova 里读写数据库要通过 Nova Conductor 以远程调用的方式来执行, 所以这里的 get_by_uuid 函数实际远程执行了 Conductor 上的相同函数来访问数据库。

我们可以从 VERSION 属性看出 Instance 对象目前的最新版本是 2.3, 之前几次版本升级分别添加了 services、keypairs 和 device_metadata 这 3 个字段。obj_make_compatible 函数的作用是根据特定的版本号把数据还原成那个版本所支持的格式。假设一个最新版本是 2.3 的 Nova 计算节点收到了一个 2.4 版本的 Instance 对象, 这超出了计算节点所能支持的版本上限, 计算节点可以通过 RPC 请求 Nova Conductor 提供 2.3 版本的数据, Nova Conductor 会通过 obj_make_compatible 函数将数据转化成计算节点理解的 2.3 版本, 这个过程称作 backport。

几乎所有的编程语言给出的第一个示例都是打印出“Hello world!”, 今天我们从另外一个角度去理解这两个单词的含义, 就是“欢迎来到虚拟机的世界”。

如果我们将 OpenStack 环境里运行在各个物理节点上的各种服务看做一系列有机的生命体, 而不是死的指令集合, 那么这就是一个虚拟机的世界。只不过与我们人类世界不同的是, 虚拟机世界里的个体是一个一个鲜活的虚拟机, 而不是人。人的世界有道德与法律去制约管理, 虚拟机的世界同样也有自己的管理结构, 这就是本章所要展示的 OpenStack 计算组件。

OpenStack 的计算组件, 也就是 Nova 项目为我们实现了这个虚拟机世界的抽象, 控制着一个个虚拟机的状态变迁与生老病死, 管理着它们的资源分配。

作为 OpenStack 中历史最为悠久的项目之一, Nova 一直都处于所有 OpenStack 项目的权力中心, 演绎着最为核心的角色, 拥有着最为广泛的关注与最多的开发者。OpenStack 基金会的用户调查也显示 Nova 一直都是部署最多的项目, 这主要是因为人们大多用 OpenStack 来部署 IaaS, 而计算组件必然是其中的核心。

OpenStack 由 NASA 与 Rackspace 发起之初, 仅有 Nova 和 Swift 两个项目, 其中 Nova 就包含了 3 个核心的领域, 即 Compute、Storage 和 Network。但随着云技术的发展, 虚拟化存储和虚拟化网络技术越来越复杂, 于是逐渐从 Nova 中分离出来, 从而有了 Neutron 与 Cinder 两个独立的项目来分别负责虚拟化网络和虚拟化存储技术。随后 Scheduler 也将独立出来, 作为一个独立的项目。最终 Nova 将缩减其项目范围, 有更加专注的目标, 这样 Nova 的复杂度将得以控制。以下是 OpenStack 官方对于 Nova 的定义:

To implement services and associated libraries to provide massively scalable, on demand, self service access to compute resources, including bare metal, virtual machines, and containers.

从中可知 Nova 将只专注于提供统一的计算资源抽象, 这些计算资源可以是物理机、虚拟机, 甚至是容器。

当然迄今为止 Nova 仍然是很复杂的项目, 为了能提供真正可用的计算资源给用户还需要与 Neutron 和 Cinder 协同工作。Nova 的代码中也包含一些历史问题, 社区开发者一直努力不断减轻这些技术债务, 消除这些历史问题, 从而让 Nova 的代码可维护性更好。

5.1 Nova 体系结构

Nova 由多个提供不同功能的独立组件组成，对外通过 REST API 通信，对内使用 RPC 进行通信，使用一个中心 DB 来存储数据，如图 5-1 所示。每个组件都可以部署一个或者多个来实现横向扩展。这样的架构也是被大部分 OpenStack 项目所采用。

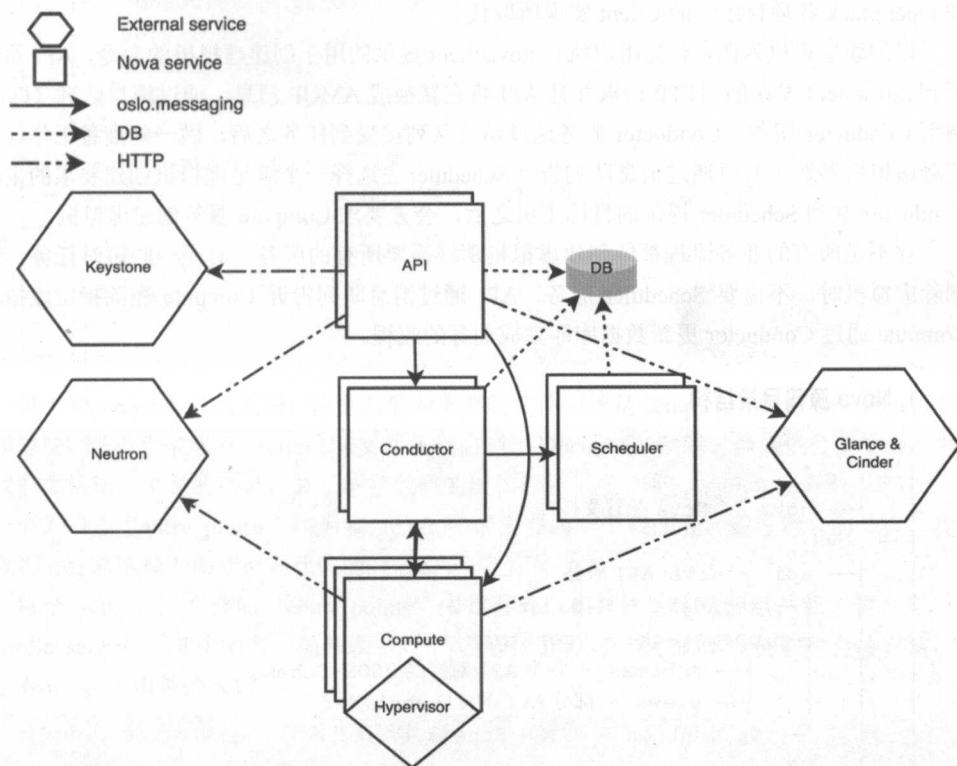


图 5-1 Nova 体系结构

由图 5-1 可以看出，目前的 Nova 主要由 API、Compute、Conductor、Scheduler 4 个核心组件所组成，它们之间通过 RPC 进行通信。

API 是进入 Nova 的 HTTP 接口，可以通过部署多个来实现横向扩展。API 依据请求是长时任务或者是短时任务，将请求发送给 Conductor 或者 Compute。长时任务请求被发送到 Conductor，Conductor 负责对其全程跟踪和调度。对于新建虚拟机或者迁移类需要调度的请求，Conductor 会向 Scheduler 请求一台符合要求的计算节点，随后 Conductor 会把请求最终发送到合适的计算节点上。Conductor 除了长时任务还负责代理其他节点的 DB 访问。这主要是为了安全问题和实现在线升级功能。最终对于虚拟机操作的请求都会发送到 Compute 组件，

Compute 负责与 Hypervisor 进行通信, 实现虚拟机的生命周期管理。对各个 Hypervisor 的支持通过 Virt Driver 框架来实现, 具体可参阅 <http://docs.openstack.org/developer/nova/support-matrix.html>。

为了简化用户对 RESTful API 的使用, Nova 提供了官方的 API 封装 python-novaclient 作为 Client, 它提供命令行供用户直接访问 Nova, 也提供了 SDK 供用户编写客户端应用程序。由于社区希望 OpenStack 对用户有统一的一致的用户体验, python-novaclient 会在将来被遗弃, 被 OpenStack 各项目统一的 Client 实现所取代。

以创建虚拟机为例, 首先用户执行 novaclient 提供的用于创建虚拟机的命令, API 服务监听到 novaclient 发送的 HTTP 请求并且 API 将它转换成 AMQP 消息, 通过消息队列 (Queue) 调用 Conductor 服务, Conductor 服务通过消息队列接受到任务之后, 做一些准备工作 (例如汇总虚拟机参数等), 再通过消息队列告诉 Scheduler 去选择一个满足虚拟机创建要求的主机, Conductor 拿到 Scheduler 提供的目标主机之后, 会去要求 Compute 服务创建虚拟机。

并不是所有的业务流程都像创建虚拟机那样需要所有的服务, 对于一些短时任务, 比如删除虚拟机时, 不需要 Scheduler 服务, API 通过消息队列告诉 Compute 删除指定虚拟机, Compute 通过 Conductor 更新数据库即完成业务的流程。

1. Nova 源码目录结构

```
.
├── etc
│   └── nova - Nova 配置文件
├── nova
│   ├── api - Nova API 服务
│   │   ├── metadata - Metadata API
│   │   ├── openstack - OpenStack API
│   │   │   ├── schemas - 各个 API 对应的 JSON-Schema
│   │   │   └── views - 部分 API 的 viewbuilder
│   │   └── validation - JSON-Schema 实现及工具
│   ├── CA
│   ├── cells - nova-cells 服务
│   ├── cert - nova-cert 服务
│   ├── cloudpipe - Cloudpipe 支持, 提供 VPN 服务
│   ├── cmd - 各个 Nova 服务的入口程序
│   ├── compute - Nova Compute 服务
│   ├── conductor - Nova Conductor 服务
│   ├── conf - 所有的配置选项
│   ├── console - nova-console 服务
│   ├── consoleauth - nova-consoleauth 服务
│   ├── db - 数据库操作
│   ├── hacking - 编码规范检查
│   └── image - Glance 接口抽象
```



```

├── ipv6 - ipv6 工具函数
├── keymgr
├── locale - 国际化相关文件
├── network - nova-network 服务
├── objects - Objects Module
├── pci - PCI/SR-IOV 支持
├── policies - 所有 policy 的默认规则
├── scheduler - Schedule 服务
├── servicegroup
├── tests - 单元测试
├── virt - Hypervisor driver
├── vnc
├── volume - Cinder 接口抽象
├── plugins - Hypervisor host 插件, 目前只有 Xenserver 一种
├── run_tests.sh
├── setup.cfg
├── setup.py
├── tools
└── tox.ini

```

对于 OpenStack 新人来说, 这里面最为重要的文件应该是 `setup.cfg`。作为 OpenStack 中的源码地图, 毫不夸张地说, `setup.cfg` 文件是我们浏览 OpenStack 代码时最为依仗的文件, 它引导我们去认识一个新的项目, 并了解它代码的结构。

而入口点 “`entry_points`” 作为 `setup.cfg` 中最最重要的一个 section, 通过对它的分析, 我们可以相对容易地找到所要研究代码的突破口。

每个 `setup.cfg` 文件的 “`entry_points`” 中都会有个相对比较特殊的组, 或者说命名空间 “`console_scripts`”, 其中的每一项都表示一个可执行的脚本, 这些脚本在部署时会被安装, 这就是 Nova 各个组件的入口。

```

console_scripts =
    nova-all = nova.cmd.all:main
    nova-api = nova.cmd.api:main
    nova-api-metadata = nova.cmd.api_metadata:main
    nova-api-os-compute = nova.cmd.api_os_compute:main
    nova-cells = nova.cmd.cells:main
    nova-cert = nova.cmd.cert:main
    nova-compute = nova.cmd.compute:main
    nova-conductor = nova.cmd.conductor:main
    nova-console = nova.cmd.console:main
    nova-consoleauth = nova.cmd.consoleauth:main
    nova-dhcpbridge = nova.cmd.dhcpbridge:main
    nova-idmapshift = nova.cmd.idmapshift:main
    nova-manage = nova.cmd.manage:main

```



```

nova-network = nova.cmd.network:main
nova-novncproxy = nova.cmd.novncproxy:main
nova-policy = nova.cmd.policy_check:main
nova-rootwrap = oslo_rootwrap.cmd:main
nova-rootwrap-daemon = oslo_rootwrap.cmd:daemon
nova-scheduler = nova.cmd.scheduler:main
nova-serialproxy = nova.cmd.serialproxy:main
nova-spicehtml5proxy = nova.cmd.spicehtml5proxy:main
nova-xvpvncproxy = nova.cmd.xvpvncproxy:main
wsgi_scripts =
nova-placement-api = nova.api.openstack.placement.wsgi:init_application

```

对于 Nova 来说,我们可以看到,除了图 5-1 所示的几个主要服务 API、Conductor、Scheduler 与 Computer, 它还提供了很多其他的服务。

- nova-all: 用于启动所有 Nova 服务的辅助脚本, 其中 API 服务作为 WSGI 服务器启动。
- nova-api: Nova 对外提供的 RESTful API 服务, 目前 Nova 共提供两种 API 服务, 即 nova-api-metadata 和 nova-api-os-compute。nova-api 根据配置文件/etc/nova/nova.conf 的 enable_apis 选项设置启动这两种服务。
- nova-api-metadata: 接受虚拟机实例 metadata (元数据) 相关的请求, 目前这部分工作由 Neutron 项目完成, 而 nova-api-metadata API 服务只有在多计算节点部署, 并且使用 nova-network 的情况下才使用。
- nova-api-os-compute: OpenStack API 服务。
- nova-cells: Cell 模块允许用户在不影响现有 OpenStack 云环境的前提下, 增强横向扩展、大规模部署能力。Cell 模块启用后, OpenStack 云环境中的主机被划分成组被称为 Cell。Cell 可以被配置成树形结构, OpenStack 云环境通过添加子 Cell 的方式进行拓展。nova-cells 负责各个 Cell 之间的通信, 以及为一个新的虚拟机实例选择合适的 Cell, 因此每个 Cell 都需要运行 nova-cells 服务。
- nova-cert: 管理 X509 证书。
- nova-compute: Compute 服务。
- nova-conductor: Conductor 服务。
- nova-console: 允许用户通过代理访问虚拟机实例 (Instance) 的控制台。已经在 G 版本中被 nova-xvpvncproxy 所取代。
- nova-novncproxy: Nova 提供了 novncproxy 代理支持用户通过 vnc 访问虚拟机。提供完整的 vnc 访问功能, 涉及几个 Nova 服务。nova-consoleauth 提供认证授权, nova-novncproxy 用于支持基于浏览器的 vnc 客户端, nova-xvpvncproxy 用于支持基于 Java 的 vnc 客户端。
- nova-dhcpbridge: 管理 nova-network 的 DHCP bridge。

- nova-manage: 提供很多与 Nova 的维护和管理相关的功能, 比如用户创建、vpn 管理等。
- nova-network: 提供网络服务, 已经被 Neutron 所取代, 目前只有在使用 Devstack 部署 OpenStack 时才会使用 nova-network。
- nova-rootwrap: 用于在 OpenStack 运行过程中以 root 身份运行某些 shell 命令。
- nova-scheduler: Scheduler 服务。

其中 nova-api 会读取 api-paste.ini, 从中加载整个 WSGI stack。最终 API 的入口点都位于 nova.api.openstack.compute 路径中。我们如果希望研究某个 API 的实现细节, 可以将这些入口点中指定的代码路径作为突破口切入进去, 从而更为有效地理清 Nova 的脉络。

Nova 中各个服务之间的通信使用了基于 AMQP 实现的 RPC 机制, 其中 nova-compute、nova-conductor 和 nova-scheduler 在启动时都会注册一个 RPC Server, 而 nova-api 因为 Nova 内部并没有服务会调用它提供的接口, 所以无需注册。下面以 nova-compute 服务为例:

```
# nova/compute/rpcapi.py

class ComputeAPI(object):

    def start_instance(self, ctxt, instance):
        version = '4.0'
        cctxt = self.router.by_instance(ctxt, instance).prepare(
            server=_compute_host(None, instance), version=version)
        # RPC cast 主要用于异步形式, 比如创建虚拟机, 在创建过程可能需要很长
        # 时间, 如果使用 RPC call 显然对性能有很大影响。cast() 的第二个参数是
        # RPC 调用的函数名, 后面的参数将作为参数被传入该函数
        cctxt.cast(ctxt, 'start_instance', instance=instance)
```

类 nova.compute.rpcapi.ComputeAPI 中的函数即为 Compute 服务提供给 RPC 调用的接口, 其他服务调用前需要首先 import 这个模块, 比如:

```
# nova/compute/api.py

def start(self, context, instance):
    LOG.debug("Going to try to start instance", instance=instance)

    instance.task_state = task_states.POWERING_ON
    instance.save(expected_task_state=[None])

    self._record_action_start(context, instance,
                              instance_actions.START)
    # 调用类 nova.compute.rpcapi.ComputeAPI 中的接口
    self.compute_rpcapi.start_instance(context, instance)
```

nova.compute.rpcapi.ComputeAPI 只是暴露给其他服务的 RPC 调用接口, Compute 服务的

RPC Server 接受到 RPC 请求后，真正完成任务的是 nova.compute.manager 模块。

```
# nova/compute/manager.py

class ComputeManager(manager.Manager):
    target = messaging.Target(version='3.35')
    @wrap_exception()
    @reverts_task_state
    @wrap_instance_event(prefix='compute')
    @wrap_instance_fault
    def start_instance(self, context, instance):
        """Starting an instance on this host."""
        .....
```

从 nova.compute.rpcapi.ComputeAPI 到 nova.compute.manager.ComputeManager 的过程即是 RPC 调用过程。

5.2 Nova API

Nova API 是访问并使用 Nova 所提供的各种服务的公共接口，作为客户端和 Nova 之间的中间层，Nova API 扮演了一个桥梁，或者说中间人的角色。Nova API 把客户端的请求传达给 Nova，待 Nova 处理完请求后再将处理结果返回给客户端。

基于这样的特殊性，Nova API 被要求保持高度的稳定，它们的名称以及返回的数据结构都不能轻易地做出改变。因此，与 Nova API 有关的 patch 都会有着非常严格的评审，任何的修改都需要一个专门的 bp (blueprint) 和 Nova Spec 进行阐释。

Nova API 自 Icehouse 开始便一直开始变革。这其中还走了一些弯路，经过几个 release 之后还是找到了最终的方向。这其中的变化可能对初学者产生了不少困惑。因此这里我们来一一描述这些 API 的变化。

- Nova v2 API: 是自 OpenStack 诞生以来，Nova 所拥有的 API。当前被标记为支持的 API，但 v2 API 已经冻结，不会再添加任何新功能，因此我们目前又称其为 Legacy V2 API。
- Nova v2.1 API: 社区所创建的新 API，在 Kilo 被标记为当前的 API，而在 Liberty 是作为默认的 API。任何新功能都会在此 API 上进行开发。
- Nova v1.1 API: 这个 API 一直是 v2 API 的一个别名，v1.1 是一个在 Nova 开放出来就已经废弃的 API，并在 Mitaka release 中已经删除了。
- Nova EC2 API: 一个 EC2 兼容 API，便于一些基于 AWS EC2 API 的应用程序可以在 OpenStack Cloud 环境中仍然可以运行。但实际情况是社区并没有太多的贡献者感兴趣维护此 API，在 Liberty 中标记为 deprecated。由一些对此兼容 API 感兴趣的贡献者创建另一个独立的项目 (<https://github.com/openstack/ec2-api>) 来基于 Nova API 实

现一个 EC2 兼容 API。

5.2.1 Nova v2.1 API

Nova v2 API 是 Nova 自诞生以来就存在的 API，但其存在的一些问题使其不能满足 OpenStack 的发展。Nova v2 API 没有提供添加新 feature 的机制，这导致开发者一直使用 Extension 作为扩展 API 的机制，从而使 Nova v2 API 拥有了大量的 Extension，而且如果继续使用这种方式，Extension 的数量会持续增长。而 Extension 机制本身也并不符合 OpenStack 的发展。随着拥有越来越多的 OpenStack 部署，不同的 OpenStack 部署通过 Extension 机制来扩展或者裁剪 API，这导致 OpenStack 完全失去了互操作性。除此之外 Nova v2 API 的框架本身还有一些问题，没有正确地处理错误的机制，这导致有些 DB 层差异被暴露在 RESTful API 当中，同样损害了 OpenStack 的互操作性。

因此 Nova v2.1 API 诞生了，Nova v2.1 API 改进了错误处理方式，覆盖掉了不同 DB 之间的差异。其中最重要的是它引入了一个新的机制来扩展 Nova API，就是 Microversion。

从此 Extension 在 v2.1 API 中将彻底被删除，用户不得再私自扩展 Nova API 和剪裁 Nova API。用户应该将他们的需求返回到社区当中，然后通过 Microversion 来实现这些需求。最终 OpenStack 将拥有一个统一的 API 来实现互操作性。

1. Microversion

Microversion 是 Nova v2.1 API 中最重要的机制。Microversion 引入了一种改变 API 的机制，而且又可以实现兼容性，由此改进了 Nova API 的互操作性。随后也有多个项目实现了此机制。

Microversion 是单调递增的，表现为 X.Y 的形式。X 从来只有在非常重大的改变并影响了整个 API 才会变化，实际上这种情况会很少发生。而其他任何 API 的改变都需要改变 Y，无论是 API 的请求，返回或是语义改变。只有 bug 才不需要 Microversion 的变化。

在 Nova 当中，任何 Nova API 的改变都需要提交 nova-specs，需要严格的 review 以防止未预期的改变破坏 API 的兼容性。

系统拥有一个最小版本和最大版本号，只要请求在这个范围之内都会被接受。最小和最大版本号可通过 version API 来查询。

```
GET /
{
  "versions": [
    {
      "id": "v2.0",
      "links": [
        {
          "href": "http://openstack.example.com/v2/",
          "rel": "self"
        }
      ]
    }
  ]
}
```

```

    ],
    "status": "SUPPORTED",
    "version": "",
    "min_version": "",
    "updated": "2011-01-21T11:33:21Z"
  },
  {
    "id": "v2.1",
    "links": [
      {
        "href": "http://openstack.example.com/v2.1/",
        "rel": "self"
      }
    ],
    "status": "CURRENT",
    "version": "2.38",
    "min_version": "2.1",
    "updated": "2013-07-23T11:33:21Z"
  }
]
}

```

从以上请求可以看出，系统中有两个 API。“id”是这个 API 的标示。可以看出“v2.0”状态是 SUPPORTED。version 和 min_version 为空，代表着这个 API 不支持 Microversion。“v2.0”就是我们所说的旧的 v2 API。“v2.1”的状态则为“CURRENT”，version 表示最大的 Microversion 版本号为“2.38”，最小的 Microversion 版本号为“2.1”。这就是 Nova 当前所支持的 API。在编写客户端程序的时候，可以通过此 version API 来识别出所访问的 API 的信息，所支持的 Microversion 范围。Microversion “2.1”是一个与 Nova v2 API 所兼容的 API。

而当客户请求的时候，可以通过发送以下 HTTP 头：

```
OpenStack-API-Version: compute 2.25
```

“compute”代表所请求的服务类型，Nova 所对应的是“compute”。“2.25”代表请求所对应的 Microversion。如果没有发送此 HTTP 头则代表是请求最小版本。如果请求超出了最大和最小版本，系统则会返回“HTTP Not Acceptable 406”。这里还有一个特殊的关键字“latest”，代表请求最新的版本。这是一个用来方便测试的关键字，在实际生产中，则需要指定请求所对应的版本号，否则在系统升级后，最新的 API 很可能已经不兼容。

5.2.2 Nova API 实现

Nova API 的代码位于 nova/api/目录下，目录结构如下：

```

.
├── metadata

```



```

├─ openstack
│   └─ compute
│       └─ schemas
│           └─ consoles.py
│           └─ flavors.py
│           └─ servers.py
│           └─ .....
│       └─ consoles.py
│       └─ extensions.py
│       └─ flavors.py
│       └─ image_metadata.py
│       └─ images.py
│       └─ __init__.py
│       └─ ips.py
│       └─ limits.py
│       └─ server_metadata.py
│       └─ servers.py
│       └─ versions.py
│       └─ .....
│       └─ views
│   └─ api_version_request.py
│   └─ extensions.py
│   └─ __init__.py
│   └─ urlmap.py
│   └─ wsgi.py
│   └─ xmlutil.py
│   └─ .....
└─ validation

```

metadata 目录下对应的是 Metadata API，这是提供给所创建的虚拟机来获得一些配置信息的 API。openstack 目录对应的是 Nova v2.1 API。为了降低维护成本，Nova v2 API 已经从 Nova 的代码中删除了。Nova v2 API 是通过在 v2.1 API 代码上执行一个兼容模式来实现的。

Nova API 是基于 WSGI 实现的。nova/api/openstack/下包含着 WSGI 基础架构的代码，其中包含一些 Nova WSGI stack 中所需要的 middleware，以及如何解析请求与分发请求的核心代码。在 nova/api/openstack/compute 中可以找到对应每个 API 的入口点。当前 Nova API 使用 JSON-Schema 来验证输入，这些 JSON-Schema 都位于 nova/api/openstack/compute/schemas/下，并使用与相应 API 所在文件相同的模块名称。对于 JSON-Schema 的验证实现则位于 nova/api/validation/。

1. Nova API 请求路由

想弄清楚 Nova API 的路由请求，可以从 Nova 如何设置这些路由请求来看。Nova 使用 Python Paste 作为工具来加载 WSGI stack。WSGI stack 通过文件 etc/nova/api-paste.ini 来配置。


```
[composite:osapi_compute]
use = call:nova.api.openstack.urlmap:urlmap_factory
/: oscomputeversions
# v21 is an exactly feature match for v2, except it has more stringent
# input validation on the wsgi surface (prevents fuzzing early on the
# API). It also provides new features via API microversions which are
# opt into for clients. Unaware clients will receive the same frozen
# v2 API feature set, but with some relaxed validation
/v2: openstack_compute_api_v21_legacy_v2_compatible
/v2.1: openstack_compute_api_v21
```

从以上配置，可以看出 Nova API 都提供了哪些 endpoints。“/” 对应的是 version API，如前面章节的讲解，可以通过这个 API 来获得所访问的 API 提供哪些版本的 API，以及 API 所支持的 Microversion 信息。“/v2” 是我们所说的 Nova v2 API，当前它通过一个兼容模式在 v2.1 代码上运行。“/v2.1” 就是 Nova 当前的 API。

```
[composite:openstack_compute_api_v21]
use = call:nova.api.auth:pipeline_factory_v21
noauth2 = cors http_proxy_to_wsgi compute_req_id faultwrap sizelimit
noauth2 osapi_compute_app_v21
keystone = cors http_proxy_to_wsgi compute_req_id faultwrap sizelimit
authtoken keystonecontext osapi_compute_app_v21
```

`nova.api.auth:pipeline_factory_v21` 是这个 API stack 的一个工厂函数，负责加载每一个 middleware。根据配置可以选择没有验证的 `noauth2` stack 或者 `Keystone` stack。`noauth2` 一般是被 `funtool` 测试来使用，在实际生产环节中，都是使用 `Keystone` 来做验证的。这里可以看到整个 stack 当中都包含了哪些 middleware。最后一个 middleware “`osapi_compute_app_v21`” 就是 v2.1 本身。在这之前的 middleware 都会对请求或返回做一些处理，比如添加请求 id 用来帮助调试，对错误返回进行统一的包装，以及对请求 token 的验证等等。

```
[app:osapi_compute_app_v21]
paste.app_factory = nova.api.openstack.compute:APIRouterV21.factory
```

最后我们就可以找到 Nova v2.1 API 的入口了。`nova.api.openstack.compute:APIRouterV21.factory` 又是一个工厂函数，用来创建 Nova v2.1 API。

`APIRouterV21` 继承自 `nova.api.openstack.APIRouterV21`，主要完成对所有资源的加载以及路由规则的创建，自此 WSGI Routes 模块开始参与进来。

```
# nova/api/openstack/__init__.py

# Router 类对 WSGI routes 模块做了简单的封装
class APIRouterV21(base_wsgi.Router):

    API_EXTENSION_NAMESPACE = 'nova.api.v21.extensions'
```

```

def __init__(self, init_only=None):

    # 使用 stevedore 的 EnabledExtensionManager 类载入位于 setup.cfg
    # 中命名空间 nova.api.v21.extensions 下的所有资源。采用
    # EnabledExtensionManager 的形式，可以在加载的时候使用 check_func
    # 函数进行检查，比如检查是否加载了重复的 API
    self.api_extension_manager =
stevedore.enabled.EnabledExtensionManager(
    namespace=self.API_EXTENSION_NAMESPACE,
    check_func=_check_load_extension,
    invoke_on_load=True,
    invoke_kwds={"extension_info": self.loaded_extension_info})

    mapper = ProjectMapper()

    self.resources = {}

    if list(self.api_extension_manager):
        # 对所有资源依次调用 _register_resources(), 这个函数会调用各个资源的
        # get_resources() 函数并进行资源注册, 同时使用 mapper 对象建立路由规则。
        self.api_extension_manager.map(self._register_resources,
                                       mapper=mapper)
        # 对所有资源依次调用 _register_resources(), 这个函数会调用各个资源的
        # get_controller_extensions() 函数扩展现有资源及其操作
        self.api_extension_manager.map(self._register_controllers)

    super(APIRouterV21, self).__init__(mapper)

```

从上面的代码可以看出来，APIRouterV21 通过 stevedore 加载各个 API 实现模块。然后 Nova 使用 Python Routes 模块作为 URL 映射的工具。APIRouterV21 将各个模块所实现的 API 对应的 URL 注册到 mapper 当中。并把每个资源都被封装成一个 nova.api.openstack.wsgi.Resource 对象。当解析每个 URL 请求的时候，可以通过 URL 映射找到 API 对应的 Resource object。

```

# nova/wsgi.py

class Router(object):

    def __init__(self, mapper):
        self.map = mapper
        # 使用 routes 模块将 mapper 与 _dispatch() 关联起来
        # routes.middleware.RoutesMiddleware 会调用 mapper.routematch()
        # 函数来获取 url 的 controller 等参数，保存在 match 中，并设置 environ

```

```

# 变量供_dispatch()使用
#     environ['wsgiorg.routing_args'] = ((url), match)
#     environ['routes.url'] = url
self._router = routes.middleware.RoutesMiddleware(self._dispatch,
                                                    self.map)

@webob.dec.wsgify(RequestClass=Request)
def __call__(self, req):
    # 根据 mapper 将请求路由到适当的 WSGI 应用, 即资源上。每个资源会在自己
    # 的 call ()方法中, 根据 HTTP 请求的 url 将其路由到对应 Controller
    # 上的方法
    return self._router

@staticmethod
@webob.dec.wsgify(RequestClass=Request)
def _dispatch(req):
    # 读取 HTTP 请求的 environ 信息并根据前面设置的 environ 找到 url 对应的
    # Controller
    match = req.environ['wsgiorg.routing_args'][1]
    if not match:
        return webob.exc.HTTPNotFound()
    app = match['controller']
    return app

```

再追溯到 nova.api.openstack.APIRouterV21 的父类, 可以看到, 请求调用 Python Routes 提供的 RoutesMiddleware 来解析之前创建的 URL mapping, 最后通过 `_dispatch` 函数回调回来, 取出其中的 Resource 对象。再调用 Resource 对象的 `__call__` 方法, 这其中做了一些 API 所需的处理, 比如 Microversion 解析, 请求数据类型的解析。最终会通过请求调用对应的 API 模块中的方法。

2. Nova API 的实现

Resource 对象会将请求的 API 映射到对应的 Controller 方法上, 以及根据请求找到对应 Microversion 的 Controller 方法。

每个 API 对应的 Controller 都在 nova/api/openstack/compute/目录下的各个 API 对应的模块中。这些模块注册就是 setup.cfg 中所描述的。

例如, keypairs API 在 setup.cfg 中注册的是 nova.api.openstack.compute.keypairs:Keypairs, 在 nova/api/openstack/compute/keypairs.py 中可以找到 Keypairs 类。

```

class Keypairs(extensions.V21APIExtensionBase):
    """Keypair Support."""

    name = "Keypairs"

```

```

alias = ALIAS
version = 1

def get_resources(self):
    resources = [
        extensions.ResourceExtension(ALIAS,
                                     KeypairController())
    ]
    return resources

def get_controller_extensions(self):
    controller = Controller()
    extension = extensions.ControllerExtension(self, 'servers',
controller)
    return [extension]

def server_create(self, server_dict, create_kwargs,
                  body_deprecated_param):
    create_kwargs['key_name'] = server_dict.get('key_name')

def get_server_create_schema(self, version):
    if version == '2.0':
        return keypairs.server_create_v20
    else:
        return keypairs.server_create

```

`keypairs` 类描述了要注册的资源或者所要扩展的资源。`get_resources` 方法用来返回所要注册的资源，这里注册了一个资源，名字为 `ALIAS` 所指定。`ALIAS` 的值是“`os-keypairs`”，所以在 API 中所对应的 URL 为“`/v2.1/os-keypairs`”。这个资源所对应的 Controller 是 `KeypairController`。除了添加新资源，还可以扩展现有资源，则通过 `get_controller_extensions` 方法来注册，这里扩展了 `servers` 资源。

首先我们先看看一个新资源的 Controller 是如何实现的。

```

class KeypairController(wsgi.Controller):

    @wsgi.Controller.api_version("2.10")
    @wsgi.response(201)
    @extensions.expected_errors((400, 403, 409))
    @validation.schema(keypairs.create_v210)
    def create(self, req, body):
        user_id = body['keypair'].get('user_id')
        return self._create(req, body, type=True, user_id=user_id)

    @wsgi.Controller.api_version("2.2", "2.9") # noqa
    @wsgi.response(201)

```



```

@extensions.expected_errors((400, 403, 409))
@validation.schema(keypairs.create_v22)
def create(self, req, body):
    return self._create(req, body, type=True)

@wsgi.Controller.api_version("2.1", "2.1") # noqa
@extensions.expected_errors((400, 403, 409))
@validation.schema(keypairs.create_v20, "2.0", "2.0")
@validation.schema(keypairs.create, "2.1", "2.1")
def create(self, req, body):
    return self._create(req, body)
.....

@wsgi.Controller.api_version("2.1", "2.1")
@wsgi.response(202)
@extensions.expected_errors(404)
def delete(self, req, id):
    self._delete(req, id)

@wsgi.Controller.api_version("2.2", "2.9") # noqa
@wsgi.response(204)
@extensions.expected_errors(404)
def delete(self, req, id):
    self._delete(req, id)

@wsgi.Controller.api_version("2.10") # noqa
@wsgi.response(204)
@extensions.expected_errors(404)
def delete(self, req, id):
    # handle optional user-id for admin only
    user_id = self._get_user_id(req)
    self._delete(req, id, user_id=user_id)
.....

@wsgi.Controller.api_version("2.10")
@extensions.expected_errors(404)
def show(self, req, id):
    # handle optional user-id for admin only
    user_id = self._get_user_id(req)
    return self._show(req, id, type=True, user_id=user_id)

@wsgi.Controller.api_version("2.2", "2.9") # noqa
@extensions.expected_errors(404)
def show(self, req, id):

```

```

        return self._show(req, id, type=True)

    @wsgi.Controller.api_version("2.1", "2.1") # noqa
    @extensions.expected_errors(404)
    def show(self, req, id):
        return self._show(req, id)
    .....

    @wsgi.Controller.api_version("2.35")
    @extensions.expected_errors(400)
    def index(self, req):
        user_id = self._get_user_id(req)
        return self._index(req, links=True, type=True, user_id=user_id)

    @wsgi.Controller.api_version("2.10", "2.34") # noqa
    @extensions.expected_errors(())
    def index(self, req):
        # handle optional user-id for admin only
        user_id = self._get_user_id(req)
        return self._index(req, type=True, user_id=user_id)

    @wsgi.Controller.api_version("2.2", "2.9") # noqa
    @extensions.expected_errors(())
    def index(self, req):
        return self._index(req, type=True)

    @wsgi.Controller.api_version("2.1", "2.1") # noqa
    @extensions.expected_errors(())
    def index(self, req):
        return self._index(req)
    .....

```

KeypairController 中，公共方法有 5 类 index/create/get/update/delete，这在 API 中对应如下。

- index: GET /v2.1/os-keypairs。
- create: POST /v2.1/os-keypairs。
- get: GET /v2.1/os-keypairs/{id}。
- update: PUT /v2.1/os-keypairs/{id}。
- delete: DELETE /v2.1/os-keypairs/{id}。

用户可以发现这 4 类方法有多个声明，这多个声明代表对应不同版本的 Microversion。方法所对应的 Microversion 通过 decorator “wsgi.Controller.api_version” 来指定。其分别对应的版本如下。

- `@wsgi.Controller.api_version("2.1", "2.1")`: Microversion 为 2.1。
- `@wsgi.Controller.api_version("2.2", "2.9")`: Microversion 为 2.2 到 2.9。
- `@wsgi.Controller.api_version("2.10")`: Microversion 为 2.10 到最新版本。

方法中其他的 decorator 也十分重要，它们分别代表的含义如下。

- `@extensions.expected_errors((400, 403, 409))`: API 所允许的错误返回码。这拦截了所有未预期的错误。
- `@validation.schema(keypairs.create_v21, "2.1", "2.1")`: Microversion 为 “2.1” 请求所对应的 JSON-Schema。
- `@wsgi.response(201)`: API 请求正常返回码。

API 的输入请求的格式验证是通过 JSON-Schema，比如 `create` 方法对应 Microversion 2.1 的 JSON-Schema，位于 `nova/api/openstack/compute/schemas/keypairs`:

```
create = {
    'type': 'object',
    'properties': {
        'keypair': {
            'type': 'object',
            'properties': {
                'name': parameter_types.name,
                'public_key': {'type': 'string'},
            },
            'required': ['name'],
            'additionalProperties': False,
        },
    },
    'required': ['keypair'],
    'additionalProperties': False,
}
```

这个 JSON-Schema 代表着接受一个字典，root 层级只接受一个 key “keypair”，而且必须出现。“keypair” 的 value 同样是一个字典，接受两个 key，即 `name` 和 `public_key`。其中，`name` 必须提供，`public_key` 为可选。不能有任何其他的 key。`public_key` 接受一个字符串。`name` 接受的类型在变量 `parameter_types.name` 中。

```
name = {
    # NOTE: Nova v2.1 API contains some 'name' parameters such
    # as keypair, server, flavor, aggregate and so on. They are
    # stored in the DB and Nova specific parameters.
    # This definition is used for all their parameters.
    'type': 'string', 'minLength': 1, 'maxLength': 255,
    'format': 'name'
}
```

同样是接受一个字符串, 最短 1 个字符, 最长 255 个字符。格式为“name”。这个“name”格式的定义又可以在 `nova/api/validation/validators` 中找到。

```
@jsonschema.FormatChecker.cls_checks('name', exception.InvalidName)
def _validate_name(instance):
    regex = parameter_types.valid_name_regex
    try:
        if re.search(regex.regex, instance):
            return True
    except TypeError:
        # The name must be string type. If instance isn't string type, the
        # TypeError will be raised at here.
        pass
    raise exception.InvalidName(reason=regex.reason)
```

这是一个 `FormatChecker` 注册到 `schema validator` 当中。可以看出, 这个实现就是定义了一个正则表达式, 然后通过正则表达式来验证这个 `Instance`。

现在可以看一个方法中如何具体实现一个 API 的功能了。其实剩下的已经很简单。通过方法的接口可以得到 `webob.Request` 对象, 从 `Request` 对象中可以获取其他请求参数, 如 HTTP 请求头和 `Query` 参数。同时对于单个资源的操作, 参数中提供了资源对应的 `id`。通过这些参数, 可以执行 API 对应的业务逻辑。最终 API 的返回也是一个字典。

除了这些标准的 CURD 方法, 还可以添加 action。从 `nova/api/openstack/compute/evacuate.py` 中可以看到如何注册 `evacuate action`:

```
@extensions.expected_errors((400, 404, 409))
@wsgi.action('evacuate')
@validation.schema(evacuate.evacuate, "2.1", "2.12")
@validation.schema(evacuate.evacuate_v214, "2.14", "2.28")
@validation.schema(evacuate.evacuate_v2_29, "2.29")
def _evacuate(self, req, id, body):
    .....
```

这里“`@wsgi.action('evacuate')`”用来标识这个方法对应的 action。

对于扩展一个资源, 可以看 `keypairs` 模块中对应的另一个 `Controller`:

```
class Controller(wsgi.Controller):

    def _add_key_name(self, req, servers):
        for server in servers:
            db_server = req.get_db_instance(server['id'])
            # server['id'] is guaranteed to be in the cache due to
            # the core API adding it in its 'show'/'detail' methods.
            server['key_name'] = db_server['key_name']
```

```

def _show(self, req, resp_obj):
    if 'server' in resp_obj.obj:
        server = resp_obj.obj['server']
        self._add_key_name(req, [server])

    @wsgi.extends
    def show(self, req, resp_obj, id):
        context = req.environ['nova.context']
        if context.can(kp_policies.BASE_POLICY_NAME, fatal=False):
            self._show(req, resp_obj)

    @wsgi.extends
    def detail(self, req, resp_obj):
        context = req.environ['nova.context']
        if 'servers' in resp_obj.obj and context.can(
            kp_policies.BASE_POLICY_NAME, fatal=False):
            servers = resp_obj.obj['servers']
            self._add_key_name(req, servers)

```

这个 Controller 是扩展 servers 资源, 想要扩展对应的方法则使用相同的方法名和 decorator “@wsgi.extends”。当原始的资源方法调用完成之后, 会调用对应的扩展方法。Keypairs 中的扩展方法就是在 servers 资源的返回中添加一个 “key_name” 字段, 用来表示 server 所使用的 key。

5.3 Rolling Upgrade

OpenStack 每 6 个月发布一个版本, 每一个版本包含大量的 bug 修复和新功能。可否持续升级 OpenStack 成为了很重要的问题。

最开始, 各个项目只能做到完全 offline 升级, 造成很长的 downtime, 这对用户来说是很糟糕的体验。随后 Nova 开始实现 Rolling Upgrade, 使得整个系统不必在一个原子的升级操作中完成。而是分步地将系统的一小部分进行升级, 通过多步升级, 使得系统 downtime 得以减少。最终这也是各个项目开始复制的功能, 得以最小的 downtime 来升级整个 OpenStack 部署。最终极的目标是实现 “Live Upgrade: Zero downtime”。

Nova 所实现的 Rolling Upgrade 基本规则如下:

- 保证数据平面没有任何 downtime, 对于 Nova 来说数据平面就是虚拟机。在整个升级过程中, 所有的 VM 可以继续保持正常的运行。对于控制平面, 尽量减少 downtime, 并且实现滚动升级。
- 仅提供 N 到 N+1 版本的升级。并不提供跨版本升级。如果需要一次升级多个版本, 必须逐版本升级到最终所需版本。

让我们看看 Nova 的 Rolling Upgrade 基本步骤和原理。

- 1) 更新控制节点上的 Nova 代码, 包括所有的 Python 依赖。此刻只是代码更新了, Nova 的服务进程并没有重启, 仍然使用内存中已经加载的旧代码运行。
- 2) 在新的代码上通过 “nova-manage db sync” 和 “nova-manage api_db sync” 升级数据库 schema。在 schema 升级之后, 新的 column 和 table 都会被添加。也就是说, 数据库 schema 已经处于 N+1 版本, 代码仍然处于 N 版本。为了保证正常工作, 数据库 schema 值会做向后兼容的改变。
- 3) 重启控制节点上的所有 Nova 服务进程。通过友好的方式让服务进程推出, 如发送 SIG_TERM 信号。这样服务进程在退出前可以完成现有的请求, 使得用户不会遇到突然的服务中断或者造成未完成的数据。服务进程重启后, 新代码将被运行。也就是所有控制节点位于 N+1 版本上, 所有计算节点仍处于 N 版本上。
- 4) 滚动升级计算节点。同样是通过友好的方式中止服务进程。此时集群中会有 N 和 N+1 版本的计算节点。N+1 版本的控制器和计算服务进程会照顾好兼容性。
- 5) 所有计算节点升级完成之后, 发送 SIG_HUP 信号给所有服务进程, 使得所有服务进程开始使用新的 rpc 接口进行通信。此时所有服务都升级完成, 并且所有的新功能也完全启用。
- 6) 运行数据在线升级。

在第 2) 步中, 只有数据库的 schema 升级了, 而并不包含数据的迁移升级。数据迁移升级是在运行时进行。大部分时候是有机会碰触某个数据时顺便进行迁移升级。只有当所有数据完成迁移升级才算最终的彻底升级完成。当用户想向再下一版本升级的时候, 需要彻底完成上一次升级。因此也提供了工具强制执行所有数据迁移, 以确保上次升级的所有数据完成迁移。

整个升级过程中, 唯一的 downtime 位于第 3) 步, 而这个 downtime 也很短暂, 只是一个服务重启的过程。

5.3.1 Rolling Upgrade 实现

实现 Rolling Upgrade 需要通过多种技术, 而其中最重要几项技术如下:

- RPC versioning。
- VersionedObject。
- Conductor。

RPC Versioning 保证了不同组件在不同版本的通信兼容性。VersionedObject 是保证了组件通信对于复杂数据的兼容性, 而 Conductor 就是为了来帮助翻译这些数据的。

1. RPC Versioning

为了保证各个组件可以在不同的版本上工作, 就要保证各个组件通信方式上兼容。Nova 各个组件的通信方式是通过消息队列, 通过对 RPC 接口的版本化来实现兼容性。

这个版本化是在各个组件的客户端接口中实现的。对于 nova-compute 的 RPC 接口，位于 nova/compute/rpcapi.py:

```
class ComputeAPI(object):
    '''Client side of the compute rpc API.

    API version history:

    * 1.0 - Initial version.
    * 1.1 - Adds get_host_uptime()
    * 1.2 - Adds check_can_live_migrate_[destination|source]
    * 1.3 - Adds change_instance_metadata()
    * 1.4 - Remove instance_uuid, add instance argument to
            reboot_instance()
    * 1.5 - Remove instance_uuid, add instance argument to
            pause_instance(), unpause_instance()
    .....
```

在 ComputeAPI 对象的开端，通过注释记录了各个版本的改动。下面以 4.7 版本为例：

```
* 4.7 - Add attachment_id argument to detach_volume()
```

日志中表示此版本为 detach_volume 接口添加了一个新的参数 attachment_id。再看看接口是如何实现兼容的。

```
def detach_volume(self, ctxt, instance, volume_id, attachment_id=None):
    extra = {'attachment_id': attachment_id}
    version = '4.7'
    client = self.router.by_instance(ctxt, instance)
    if not client.can_send_version(version):
        version = '4.0'
        # 当发现当前的RPC通信版本不支持最新版本,则将attachment_id弹出参数列表。
        extra.pop('attachment_id')
    cctxt = client.prepare(server=_compute_host(None, instance),
                           version=version)
    cctxt.cast(ctxt, 'detach_volume',
               instance=instance, volume_id=volume_id, **extra)
```

可以看到，如果当前的 RPC Version Pinning 不是在 4.7 及以上版本的时候，新的参数 attachment_id 会被弹出参数列表。也就是说被调用端并不会看到这个新参数，因此被调用端处于旧的版本是仍然可以继续工作的，不会因为未知的新参数而发生错误。

2. Conductor

Conductor 服务 nova-conductor 最初于 Grizzly 版本中发布，作为 Nova 项目的核心模块之一，它在整个 Nova 中相当于组织者的角色，主要提供了 3 项基础功能。

由于 Nova conductor 连接了 nova-api、nova-compute 和 nova-scheduler 服务，所以它首先提供了长时任务编排（task orchestration）功能。Nova 将所有耗时长，跨节点，易错但相对固定的处理流程抽象成任务，包括虚拟机启动、热迁移、冷迁移等。Conductor 作为任务的组织者，不仅能对同时进行的多个任务的状态进行跟踪，还能完成错误处理、恢复等一系列功能。

其次，Conductor 为 nova-compute 提供了数据库的代理访问机制，它不仅是数据库访问的一层安全保障，还在数据库升级过程中为旧的 nova-compute 节点提供了向下的兼容性。

在模块间通信的过程中，由于不同的服务可能运行在不同的代码版本上，Conductor 为传输的数据对象提供了版本兼容功能，让处于不同版本的 Nova 服务能够识别 RPC 请求中数据对象的版本，并完成不兼容的对象自动转换。

由于 Conductor 服务本身是无状态的，用户可以在运行过程中任意调整 Conductor 的数量和位置。在性能和稳定性要求高的部署环境中，用户可以非常容易地对 Conductor 服务实现高可用和性能的横向扩展。

Conductor 服务的源码位于 nova/conductor 目录：

```
.
├── api.py
├── manager.py
├── rpcapi.py
├── tasks - 任务管理代码
│   ├── live_migrate.py
│   └── migrate.py
```

一般来说，rpcapi.py 文件与 RPC 相关，其他服务将这个模块导入就可以使用它提供的接口远程调用 nova-conductor 提供的服务，nova-conductor 注册的 RPC Server 接收到 RPC 请求后，再由 manager.py 文件中的类 ComputeTaskManager 和 ConductorManager 来完成任务的编排，以及数据库访问的代理。但是由于 Conductor 服务访问的特殊性，api.py 文件中又对接口的调用做了一层封装，其他模块需要导入的是 api 模块，而不是 rpcapi 模块。

数据库代理为 nova-compute 服务的数据库访问提供一层额外的安全保障。在此之前，nova-compute 都是直接访问数据库，一旦其被攻击，则数据库会面临直接暴露的危险。此外，代理机制也使得 nova-computer 与数据库解耦，因此在保证 Conductor API 兼容性的前提下，数据库 schema 升级的同时并不需要也去升级 nova-computer。

由于 Conductor 的横向扩展能力，nova-compute 对数据库访问的性能也有相应的保障。此前当使用协程时，Compute 服务的所有的数据库访问都是阻塞的。在引入 nova-conductor 之后，nova-compute 就可以通过创建多个协程通过 nova-conductor 适用非阻塞的 RPC 协议来访问数据库。当然这么做也不可避免有一些限制，RPC 调用是有延时的，nova-conductor 本身访问数据库也是阻塞的，当部署的 nova-conductor 实例过少时，阻塞和延时会更加突出性能的问题。

目前为止，如图 5-2 所示，nova-compute 所有访问数据库的动作都要通过 Nova Objects 交给 nova-conductor 来完成。出于安全性考虑，nova-conductor 和 nova-compute 不能部署在同

一服务器上。

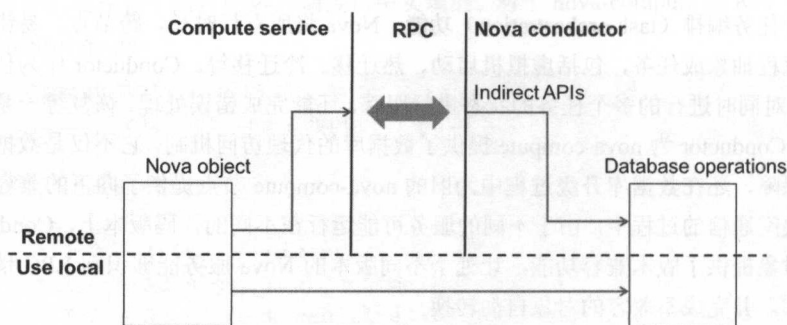


图 5-2 数据库代理

在服务启动时, Nova 的其他组件通过 `nova.conductor.API` 类使用 RPC 请求通过 Conductor 间接访问数据库。nova-compute 通常会部署在多个独立的物理主机中, 代码升级往往会导致多个 Compute 服务运行在不同的版本上, 这就需要 Conductor 的数据库代理功能为旧版本的 Compute 服务提供兼容性支持。非 nova-compute 服务, 如 nova-api 和 nova-scheduler, 则没有数据库代理功能。这是因为非 Compute 服务往往会统一部署在控制节点中, 可以一并升级, 而且同时直接访问数据库可以降低 Conductor 的压力。

Nova 子服务对数据库的访问, 无论是本地访问或者通过 RPC 远程调用, 最终真正完成数据库操作的都是 `conductor/manager.py` 里的类 `ConductorManager`。由于 Nova 中几乎所有的数据库操作都被封装在对应 Nova Object 的类方法和对象方法中, `ConductorManager` 只需要实现 `oslo_versionedobjects` 所定义的 `VersionedObjectIndirectionAPI` 即可作为服务端提供代理服务。实现的接口包括 `object_action`, `object_class_action_versions` 和 `object_backport_versions`, 分别用于 `VersionedObject` 内部关于远程对象方法, 类方法及对象版本兼容的服务端实现。

关于为旧 Compute 服务数据库访问的兼容功能, Nova 是通过 RPC 服务中实现的对象版本自动兼容来实现的。它主要对应着 Conductor 的 `object_backport_versions` 接口并依赖 `oslo_messaging` 对 `oslo_versionedobjects` 序列化功能的支持, 具体的实现见下一节内容。

3. Versioned Object Model

Versioned Object Model 由 Redhat 的 Dan Smith 提出, 在 Icehouse 版本中开始添加, 在 Juno 版本中基本实现了所有的功能。

Versioned Object Model 应该说是 Nova 中数据管理方式的分水岭, 在此之前, 对每一个数据库表的操作都放在同一个文件里, 比如 `flavor.py`, 使用时直接调用这个文件中的函数去修改数据库。而 Object Model 引入后, 新建了 Flavor 对象与 flavor 表相对应, 将对 flavor 表的操作都封装在 Flavor 对象里, 需要通过 Flavo 对象的函数去进行数据库操作。

Versioned Object Model 的引入主要实现了如下的功能:

- Nova 数据库访问方式与对象构建过程的解耦: 在 Object Model 支持远程数据库访问

的同时，也能支持已有的本地直接访问数据库功能。用户可以通过服务配置项切换数据库访问方式，而且开发者无需对任何一种访问方式实现额外的兼容代码。

- nova-compute 和数据库的在线升级：之前数据库的内容有所变动并进行升级时，必须对 nova-compute 也做相应的更新并升级。Object Model 引入后，每个对象都会维护一个版本号，RPC 请求里会包括这个版本号，数据库内容升级后，如果 nova-compute 没有升级仍然请求旧的版本，nova-conductor 将会把数据封装成旧的版本返回给 nova-compute。

比如，nova-compute 节点上对象的版本是 1.1，而 nova-conductor 节点上对象的版本是 1.2，版本 1.2 中新增了一个变量 new_value，当 nova-compute 发送 RPC 请求给 nova-conductor 时，nova-conductor 会根据 1.1 版本的消息生成新的对象，而将 new_value 赋值为 None。

- 对象属性类型的声明：Python 的一大特色就是无需声明变量类型，Python 可以自动判断，但是像 MySQL 这样的数据库就没这么智能了，所以经常发生把 int 当成 str 存进数据库的情况，为了减少这方面的 bug，对象的属性就需要明确声明自己的类型。
- 减少写入数据库的数据量：每次修改数据库中的表，或者 nova-compute 更新对象属性时，只需要进行增量更新，并不用将整个对象所有的属性都更新一遍，每个对象都有一个 _change_field 属性用来记录变化的量，从而减少写入数据库的数据量。

Object Model 并不是一个单独的服务，它使用了面向对象的思想对数据进行了封装，并为封装的数据提供了数据库代理、RPC 版本兼容、流量优化等一系列高级特性。

Object Model 代码位于 nova/objects 目录，里面的每一个类都对应数据库中的一个表，比如类 ComputeNode 对应了数据库的 compute_nodes 表。

```
# nova/objects/compute_node.py

# 基类 base.NovaObject 中会记录变化的字段，在更新数据库时只更新这些变化的字段
class ComputeNode(base.NovaPersistentObject, base.NovaObject):
    # Version 1.0: Initial version
    # Version 1.1: Added get_by_service_id()
    # Version 1.2: String attributes updated to support unicode
    # Version 1.3: Added stats field
    # Version 1.4: Added host ip field
    # Version 1.5: Added num_topology field
    # .....
    # Version 1.16: Added disk_allocation_ratio

    # VERSION 是 ComputeNode 对象的版本号，当添加或删除下面 fields 字典里
    # 的 key 或类 ComputeNode 中的方法时都必须增加版本号

    VERSION = '1.16'
```

```

# 字典 fields 是 ComputeNode 对象所维护的信息，这个字典的值并不一定包含
# ComputeNode 表内所有信息。每一个值的类型都是 nova.object.fields 模块中定义
# 一个类型，当对其赋值时，传入的数据类型不匹配，就会抛出异常。fields 支持
# 的类型有 Integer、Bool 和 Object 等

fields = {
    'id': fields.IntegerField(read_only=True),
    'uuid': fields.UUIDField(read_only=True),
    'service_id': fields.IntegerField(nullable=True),
    'host': fields.StringField(nullable=True),
    'vcpus': fields.IntegerField(),
    'memory_mb': fields.IntegerField(),
    'local_gb': fields.IntegerField(),
    'vcpus_used': fields.IntegerField(),
    'memory_mb_used': fields.IntegerField(),
    'local_gb_used': fields.IntegerField(),
    'hypervisor_type': fields.StringField(),
    'hypervisor_version': fields.IntegerField(),
    'hypervisor_hostname': fields.StringField(nullable=True),
    'free_ram_mb': fields.IntegerField(nullable=True),
    'free_disk_gb': fields.IntegerField(nullable=True),
    'current_workload': fields.IntegerField(nullable=True),
    'running_vms': fields.IntegerField(nullable=True),
    'cpu_info': fields.StringField(nullable=True),
    'disk_available_least': fields.IntegerField(nullable=True),
    'metrics': fields.StringField(nullable=True),
    'stats': fields.DictOfNullableStringsField(nullable=True),
    'host_ip': fields.IPAddressField(nullable=True),
    'numa_topology': fields.StringField(nullable=True),
    'supported_hv_specs': fields.ListOfObjectsField('HVSPEC'),
    'pci_device_pools': fields.ObjectField('PciDevicePoolList',
nullable=True),
    'cpu_allocation_ratio': fields.FloatField(),
    'ram_allocation_ratio': fields.FloatField(),
    'disk_allocation_ratio': fields.FloatField(),
}

def obj_make_compatible(self, primitive, target_version):
    .....

@base.remotable_classmethod
def get_by_id(cls, context, compute_id):
    db_compute = db.compute_node_get(context, compute_id)
    return cls._from_db_object(context, cls(), db_compute)
.....

```

```
@base.remotable
def destroy(self):
    db.compute_node_delete(self._context, self.id)
```

nova.objects.base 中定义有两个非常重要修饰符函数:remotable_classmethod 和 remotable,前者用于修饰类的方法,后者用于修饰类实例的方法。这两个方法由 oslo_versionedobjects 提供实现,会自动调用 VersionedObject 类加载的 indirection_api 来实现对数据库的远程调用。例如在 nova-compute 服务的启动过程中,会将 ConductorAPI 加载至 NovaObject 类中。

```
# nova/cmd/compute.py
def main():
    .....
    if not CONF.conductor.use_local:
        cmd_common.block_db_access('nova-compute')
        objects_base.NovaObject.indirection_api = \
            conductor_rpcapi.ConductorAPI()
    else:
        LOG.warning(_LW('Conductor local mode is deprecated and will '
            'be removed in a subsequent release'))
    .....

```

为了做到不同代码版本的 Nova 服务之间能够互相兼容, Nova 为消息通信实现了特制的序列化工具类 NovaObjectSerializer,并在服务启动时加载到 RPC Server 中。该类不仅能够将 Nova Objects 与最基础的字典对象实现互相转换,使之能够以字符串的形式在网络中传输,还能在发现客户端服务端要求的数据格式版本不一致时,与 Conductor 服务通信自动完成兼容。

图 5-3 所示为整个兼容过程的实现。当 Object a 需要从 Service A 传输给 Service B 时, RPC Server 会调用 NovaObjectSerializer 的 serialize_entity 方法将其转换成最简单的字典对象,并以字符串形式发送至 Service B。NovaObjectSerializer 会在 B 端尝试将字典对象恢复为同样的 Object a,但由于 Service A 与 B 不一定运行在同一个代码版本上,它们支持的数据格式也不一定一致。NovaObjectSerializer 能将字典对象中的 nova_object.name 和 nova_object.version 值与当前代码对应 Nova Object 类的版本相比较,来确定数据格式是否兼容。通常来说,当主版本号相同,且次版本号新于字典对象的次版本时,Serializer 就会认定版本兼容,并将字典内容恢复为 Nova Object。否则当版本不兼容时,Serializer 会调用 Conductor 的 object_backport_versions 方法将 Object 调整为与 Service B 兼容的版本。Conductor 能够为所有 Nova 子服务实现对象版本兼容的原因是在进行代码升级时,Conductor 服务总是应该保证是第一个更新的,这样 Conductor 就一定具有最新的 Nova Object 的实现和对应的兼容逻辑,所以能够为所有其他的服务做数据兼容。

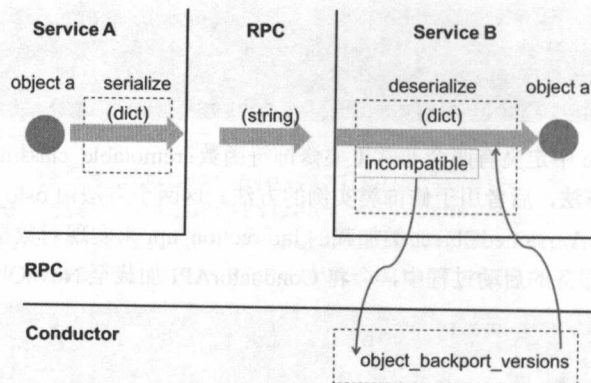


图 5-3 对象版本兼容

在进行版本兼容时，Conductor 会调用对应 Nova Object 的 `obj_make_compatible` 方法来实现数据内容的调整。例如 Instance 对象，它的兼容方法就根据传入的 `target_version` 参数来控制 `primitive` 字典的向下兼容：

```
# nova/objects/instance.py

def obj_make_compatible(self, primitive, target_version):
    super(Instance, self).obj_make_compatible(primitive, target_version)
    target_version = versionutils.convert_version_to_tuple(target_version)
    if target_version < (2, 3) and 'device_metadata' in primitive:
        del primitive['device_metadata']
    if target_version < (2, 2) and 'keypairs' in primitive:
        del primitive['keypairs']
    if target_version < (2, 1) and 'services' in primitive:
        del primitive['services']
    .....
```

5.4 Scheduler

我们人类世界所有人都在分享同一个地球，而虚拟机世界里的则是多个虚拟机分享一个或者多个主机。既然是分享而不是独占，就必须使用某种规则来进行协调。如果采用了不好的规则，某些个体就会占有过多资源，相对地，其他个体就会因缺少资源而无法生存。

虚拟机世界里，由 Scheduler（调度器）服务 `nova-schedduler` 来裁决虚拟机的生存空间与资源分配，即是否有一个主机能够容纳新的虚拟机，它会通过各种规则，考虑包括内存使用率、CPU 负载等多种生存因素为虚拟机选择一个合适的主机。

类似于 `nova-volume` 被剥离成为 `Cinder`，以及 `nova-network` 被剥离为 `Neutron`，从 `Ocata`

开始社区也在致力于剥离 nova-scheduler 为独立的 Placement 服务，从而提供一个通用的调度服务被多个项目使用。

5.4.1 调度器

选择一个虚拟机在哪个主机上运行的调度方式有很多种，目前 Nova 中实现的可以在 setup.cfg 文件中找到以下几个调度器。

- **FilterScheduler**（过滤调度器）：默认载入的调度器，根据指定的过滤条件以及权重挑选最佳节点。
- **CachingScheduler**：与 FilterScheduler 功能相同，在其基础上将主机资源信息缓存在本地内存中，然后通过后台的定时任务定时从数据库中获取最新的主机资源信息。
- **ChanceScheduler**（随机调度器）：从所有 nova-compute 服务正常运行的节点中随机选择。
- **FakeScheduler**（伪调度器）：用于单元测试，没有任何实际功能的调度器。

为了便于扩展，Nova 将一个调度器必须要实现的接口提取出来成为 nova.scheduler.driver.Scheduler，只要继承虚类 Scheduler 并实现其中的接口，就可以实现一个自己的调度器。

不同的调度器并不能够共存，需要在/etc/nova/nova.conf 中通过 scheduler_driver 选项指定，默认使用和最广泛使用的是 FilterScheduler。

```
[default]
scheduler_driver = filter_scheduler
```

FilterScheduler 的工作流程如图 5-4 所示。

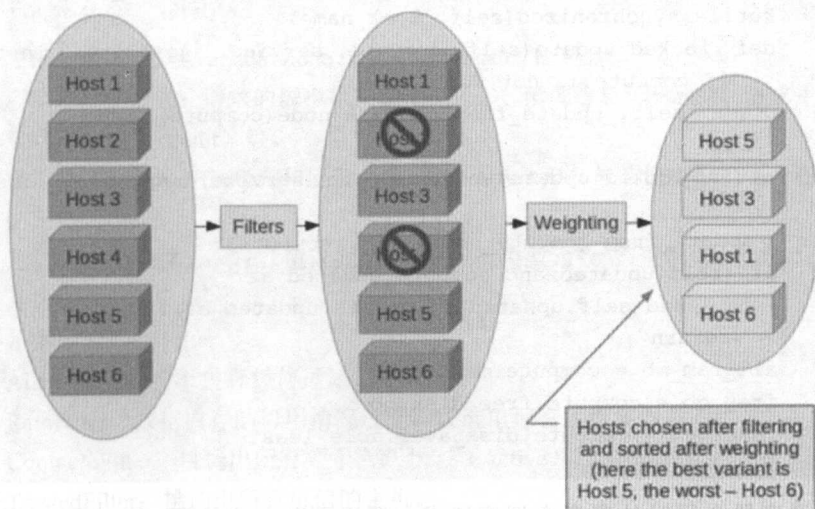


图 5-4 FilterScheduler 工作流程

FilterScheduler 首先使用指定的 Filters（过滤器）得到符合条件的主机，比如可用内存大于 2GB，然后通过配置的 Weights 对得到的主机列表计算权重并排序，获得最佳的一个。完整来说这个过程可以分为几个阶段：从 nova.scheduler.rpcapi.SchedulerAPI 发出 RPC 请求到 nova.scheduler.manager.SchedulerManager；从 SchedulerManager 到调度器实例（继承自 driver.Scheduler）；调度器从数据库中刷新缓存；使用过滤器选择符合要求的主机列表；最后使用 Weighers 为最终主机列表计算权重并排序。

从 SchedulerAPI 到 SchedulerManager 的 RPC 请求过程与 Nova 其他服务均保持一致，从 SchedulerManager 到 driver.Scheduler 的过程，会在类 SchedulerManager 初始化的时候根据配置文件的指定初始化相应的调度器。因此我们重点介绍接下来的 3 个阶段，也是 FilterScheduler 工作的核心，即缓存更新、Filtering（过滤）与 Weighting（权重计算与排序）。

1. 调度器缓存更新

nova-scheduler 在进行调度决策前需要从数据库中得到各个主机的资源数据，这些数据的收集与存储都由 nova-compute 负责。nova-compute 对数据的更新是即时更新到数据库的，并有周期性任务保证资源数据的准确性。同时，由于 nova-scheduler 无法更新数据库，所以在选择最佳主机时，要在内存中保存先前决策情况，这是通过在调度器内存中单独维护了一份缓存实现的。缓存里面包含了最近一次读取的数据库情况以及最近调度器决策导致的资源变化。这部分工作是由 nova.scheduler.host_manager.HostState 完成的。

```
# nova/scheduler/host_manager.py

class HostState(object):
    def update(self, compute, service, aggregates, inst_dict):
        @utils.synchronized(self._lock_name)
        def _locked_update(self, compute, service, aggregates, inst_dict):
            if compute is not None:
                self._update_from_compute_node(compute)
            .....
        return _locked_update(self, compute, service, aggregates, inst_dict)

    def _update_from_compute_node(self, compute):
        if (self.updated and compute.updated_at
            and self.updated > compute.updated_at):
            return
        all_ram_mb = compute.memory_mb
        free_gb = compute.free_disk_gb
        least_gb = compute.disk_available_least
        .....
```

由于调度器会同时处理多个调度请求，所以更新共享资源 HostState 时需要加锁（@utils.synchronized）来保证数据的一致性。HostState 会从数据库和缓存中更新主机数据

(compute)，服务状态 (service)，主机聚合 / 分组信息 (aggregates) 和所有相关的虚拟机状态 (inst_dict)。

在更新主机数据时，如果数据库中某个主机数据的更新时间 “updated_at” 小于 nova-scheduler 所维护数据的更新时间 (self.updated)，则说明该条数据已经过时了，此时不需要从数据库中更新。这样简单的假定可能导致数据库虽然有更新了，但由于更新时间小于缓存时间，结果没有被应用于缓存。但是在另一方面，这也保证了基于缓存的决策影响不被丢失，也是现有架构下最合适的更新机制了。关于这种实现方式对调度决策的具体影响，可以参考 <https://bugs.launchpad.net/nova/+bug/1341420/comments/24> 中的描述。

显然，为了保持自己所维护缓存的准确性，nova-scheduler 在为一个虚拟机启动请求作出决策后，它都要将其更新并从主机可用的资源中去掉虚拟机使用的部分，同时更新 self.updated 时间。

2. Filtering

Filtering 就是使用配置文件指定的各种 Filters 去过滤掉不符合条件的主机。这个阶段首先要做的一件事情是根据各个主机当前可用的资源情况，过滤掉那些不能满足虚拟机要求的主机，比如内存的容量等。

在调度器缓存更新后，各个 Filter 便粉墨登场了。Ocata 中 Nova 支持的 Filter 共有 27 个，能够处理各类信息，具体内容如下。

- 主机可用资源：内存、磁盘、CPU、PCI 设备和 NUMA 拓扑等。
- 主机类型：虚拟机类型及版本，CPU 类型及指令集等。
- 主机状态：主机是否处于活动状态、CPU 使用率、虚拟机启动数量、繁忙程度、是否可信等。
- 主机分组情况：Available Zone、Host Aggregates 信息。
- 启动请求的参数：请求的虚拟机类型 (flavor)、镜像信息 (image)、请求重试次数、启动提示信息 (hint) 等。
- 虚拟机亲合性 (affinity) 和反亲合性 (anti-affinity)：与其他虚拟机是否在同一主机上。
- 元数据处理：主机元数据、镜像元数据、虚拟机类型元数据、主机聚合 (Host Aggregates) 元数据。

下面是常用的一些 Filter。

- AllHostsFilter：不进行任何过滤。
- RamFilter：依据内存的可用情况过滤，挑选出拥有足够多内存的主机。
- ComputeFilter：挑选出所有处于活跃状态 (Active) 的主机。
- TrustedFilter：挑选出所有可信的主机。
- PciPassthroughFilter：挑选出提供 PCI SR-IOV 支持的主机。

所有的 Filter 实现都位于 nova/scheduler/filters 目录，每个 Filter 都继承自 nova.scheduler.filters.BaseHostFilter。

```
# nova/scheduler/filters/__init__.py

class BaseHostFilter(filters.BaseFilter):
    def _filter_one(self, obj, filter_properties):
        return self.host_passes(obj, filter_properties)

    def host_passes(self, host_state, filter_properties):
        raise NotImplementedError()
```

我们自己也可以很方便地通过继承类 BaseHostFilter 来创建一个新的 Filter，新建的 Filter 只需实现一个函数 host_passes()，返回结果只有两种，即满足条件返回 True，否则返回 False。

不同的 Filter 能够共存，比如配置了 Filter1、Filter2 和 Filter3，并且有 3 个主机 Host1、Host2 和 Host3，按照 Filter 的顺序，假如 Host1 和 Host3 通过了 Filter1，那么调用 Filter2 时，只考虑 Host1 和 Host3，接下来如果只有 Host1 通过了 Filter2，则调用 Filter3 时，只用考虑 Host1。

具体使用哪些 Filter 需要在配置文件中指定：

```
[default]
scheduler_available_filters= nova.scheduler.filters.all_filters
scheduler_default_filters=RetryFilter,AvailabilityZoneFilter,RamFilter,
DiskFilter,ComputeFilter,ComputeCapabilitiesFilter,ImagePropertiesFilter,
ServerGroupAntiAffinityFilter, ServerGroupAffinityFilter
```

其中，scheduler_available_filters 用于指定所有可用的 Filters；scheduler_default_filters 则表示对于可用的 Filter，nova-scheduler 默认会使用哪些。

3. Weighting

Weighting 是指对所有符合条件的主机计算权重 (Weight) 并排序，从而得出最佳的一个。

经过各种过滤器过滤之后，会得到一个最终的主机列表，保存了所有通过指定过滤器的主机。由于列表中可能存在多个主机，所以调度器还需要在他们当中选择最优的一个。类似于 Filtering，这个过程需要调用指定的各种 Weigher 模块，得出每个主机总的权重值。

所有的 Weigher 实现都位于 nova/scheduler/weights 目录，比如 RAMWeigher：

```
class RAMWeigher(weights.BaseHostWeigher):
    # 可以设置 maxval 与 minval 属性指明权重的最大值和最小值
    minval = 0

    # 权重的系数，最终排序时需要将每种 Weigher 得到的权重分别乘上它对应的这个
    # 系数，有多个 Weigher 时才有意义。对于 RAMWeigher，这个值可通过配置选项
    # ram_weight_multiplier 进行指定，默认为 1.0
```

```
def weight_multiplier(self):
    return CONF.ram_weight_multiplier

# 计算权重值，对于 RAMWeigher 仅仅返回可用内存的大小
def _weigh_object(self, host_state, weight_properties):
    return host_state.free_ram_mb
```

4. 其他调度器实现

CachingScheduler 的主要功能与 FilterScheduler 类似，唯一的区别是调度器缓存的更新时机不是在每次请求处理过程中，而是由一个额外的定时任务完成，这样避免了在大规模部署环境中过于频繁的请求导致的数据库读瓶颈。

但是，由于 nova-scheduler 对所维护的缓存数据并不会同步到数据库，它只会从数据库同步数据。如果在两次同步之间数据库的内容发生了改变，则 nova-scheduler 所维护的缓存会有一些的误差，导致了决策的失准。比如，在虚拟机销毁时，nova-compute 会去更新数据库，但因为这个过程并不会告知 nova-scheduler，它不能感知这一动作并更新自己所维护的资源数据去反映虚拟机删除所带来的资源变化。显然，由于缓存是调度器私有的，多个调度器同时运行的情况会为决策准确性带来更加严重的影响。

ChangeScheduler 的实现非常简单，没有任何缓存、过滤及权重操作，其决策过程就是随机选择一个处于正常服务状态的主机。

关于调度器的选择，需要考虑其本身的特点，包括决策精确度（FilterScheduler 优于 CachingScheduler，远优于 ChanceScheduler），同时也要考虑集群规模对调度性能的不同影响（一般来说 ChanceScheduler 优于 CachingScheduler，CachingScheduler 优于 FilterScheduler），还要考虑数据中心自身的需求。

关于具体调度器数量、种类，请求类型和集群规模对性能和决策准确度的影响，可参考 <https://www.openstack.org/assets/presentation-media/7129-Dive-into-nova-scheduler-performance-summit.pdf> 中的性能分析数据。

5.4.2 Resource Tracker

nova-compute 需要在数据库中更新主机的资源使用情况，包括内存、CPU、磁盘等，以便 nova-scheduler 获取作为选择主机的依据，这就要求每创建、迁移、删除一个虚拟机，都要更新数据库中的相关的内容。

Nova 使用 ComputeNode 对象保存计算节点的配置信息以及资源使用状况。nova-compute 服务在启动时会为当前主机创建一个 ResourceTracker 对象，其主要任务就是监视本机资源变化，并更新 ComputeNode 对象在数据库中对应的表 “compute_nodes”。

nova-compute 服务通过两种途径来更新当前主机对应的 ComputeNode 数据库记录，一是 Resource Tracker 的 Claim 机制，二是使用周期性任务（Periodic Task）。

1. 使用 Claim 机制

当一台主机被 nova-scheduler 的多个决策同时选中并发送创建虚拟机的请求时，这台主机并不一定有足够的资源来满足这些虚拟机的创建要求。Claim 机制即是在创建之前预先测试一下主机的可用资源是否能够满足新建虚拟机的需要。如果满足，则更新数据库，将虚拟机申请的资源从主机可用的资源中减掉；如果后来创建失败时，会通过 Claim 还原之前减掉的部分。

```
# nova/compute/resource_tracker.py

from nova.compute import claims

@utils.synchronized(COMPUTE_RESOURCE_SEMAPHORE)
def instance_claim(self, context, instance, limits=None):
    .....
    # 如果 Claim 返回为 None，即主机的可用资源满足不了新建虚拟机的需求
    # 则 resource tracker 不会减去 instance 占用的资源并抛出
    # ComputeResourcesUnavailable 异常。如果在 claim 成功后，虚拟机
    # 创建过程中失败（检测到任何异常），则会调用 __exit__() 方法将占用的
    # 资源返还到主机的可用资源中
    claim = claims.Claim(context, instance, self, self.compute_node,
                          pci_requests, overhead=overhead, limits=limits)

    # 通过 Conductor 更新 Instance 的 host、node 与 launched_on 属性
    self._set_instance_host_and_node(instance)
    # 根据新建虚拟机的需求计算主机可用的资源
    self._update_usage_from_instance(instance)
    elevated = context.elevated()
    # 根据最新的计算的结果更新数据库
    self._update(elevated)
```

2. 使用 Periodic Task

在 nova.compute.manager.ComputeManager 类中有个周期性任务 update_available_resource() 用于更新主机的资源数据。

```
# nova/compute/manager.py

class ComputeManager(manager.Manager):
    # 修饰符 periodic_task 表示此函数是一个周期性任务，会被周期性的调用
    @periodic_task.periodic_task(spacing=CONF.update_resources_interval)
    def update_available_resource(self, context):
        compute_nodes_in_db = self._get_compute_nodes_in_db(context,
                                                                use_slave=True)
        nodenames = set(self.driver.get_available_nodes())
```



```

# 更新所有主机数据库中的资源数据
for nodename in nodenames:
    self.update_available_resource_for_node(context, nodename)

self._resource_tracker_dict = {
    k: v for k, v in self._resource_tracker_dict.items()
    if k in nodenames}

for cn in compute_nodes_in_db:
    if cn.hypervisor_hostname not in nodenames:
        LOG.info(_LI("Deleting orphan compute node %s"), cn.id)
        cn.destroy()

```

两种更新途径并不冲突，Claim 机制是在每次主机资源消耗发生变化时更新，能够保证数据库里的可用资源及时更新，以便为 nova-scheduler 提供最新的数据。周期性更新是为了保证数据库内信息的准确性，它每次都会通过 Hypervisor 重新获取主机的资源信息，并将这些信息更新到数据库中。

5.4.3 调度流程

如果将 Nova 比做一个生物体，那调度在 Nova 中的作用相当于其神经系统，它会根据变化的环境（主机的组织结构、状态以及其使用情况）和刺激（虚拟机启动请求）作出决策（调度决定），并作出一系列行为反应（在选择的主机上启动虚拟机，并保存调度结果），而调度器在这个庞大系统中相当于神经中枢。要了解完整的调度是怎样完成的，我们需要对整个调度子系统有基本的了解。

如图 5-5 所示，整个调度子系统主要由 Nova 的四大子服务组成，即 nova-api、nova-conductor、nova-scheduler 和 nova-compute 服务。当用户发起一个新的请求时，该请求首先在 nova-api 中处理。nova-api 会对请求做一系列检查，包括请求是否合法、配额是否足够、是否有符合要求的网络、镜像及虚拟机类型等。当检查通过后，nova-api 就会为该请求分配一个唯一的虚拟机 ID，并在数据库中新建对应的项来记录虚拟机的状态。然后，nova-api 将请求发送给 nova-conductor 处理。

nova-conductor 作为一个协调者角色，主要管理服务之间的通信和任务处理。它在接收到请求之后，会为 nova-scheduler 创建一个 RequestSpec 对象，用来包装调度相关的所有请求资料，然后远程调用 nova-scheduler 服务的 select_destination 接口。

nova-scheduler 则会通过接收到的 RequestSpec 对象，根据数据库中最系统的状态作出调度决定，告诉 nova-conductor 把该请求调度到合适的计算节点上。nova-conductor 在得知调度器的决定后，会把请求发送到对应的 nova-compute 服务。

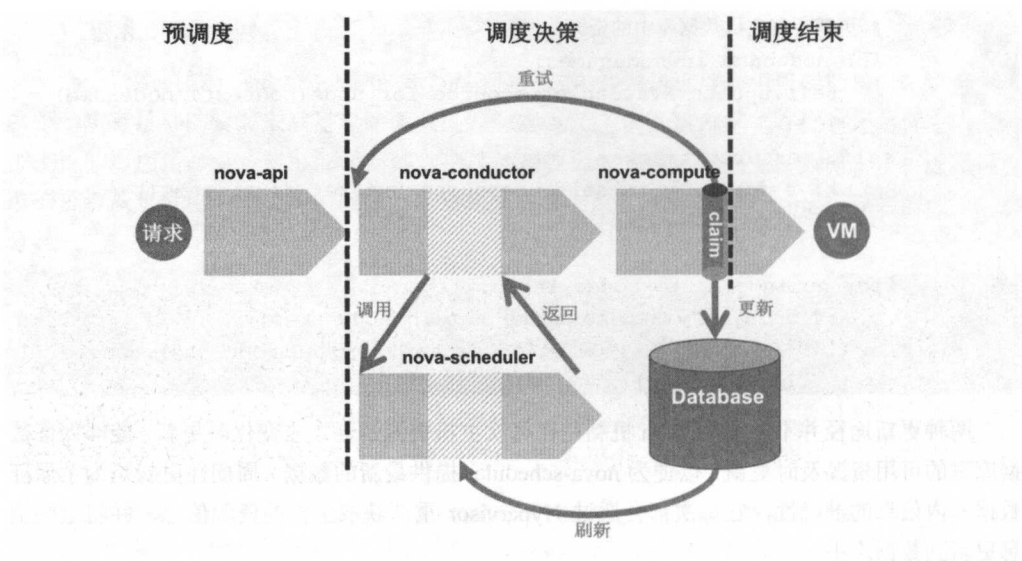


图 5-5 调度子系统

每个 nova-compute 服务都运行有独立的资源监视器 (Resource Tracker) 来监视本地主机的资源使用情况。当计算节点接收到请求时，资源监视器能够检查主机是否有足够的对应资源。如果资源足够，nova-compute 就会允许在当前主机中启动请求所要求的虚拟机，并在数据库中更新最新的虚拟机状态，同时将最新的主机资源情况更新到数据库中。若当前主机不符合请求的资源要求时，nova-compute 则会拒绝启动虚拟机，并将请求重新发送到 nova-conductor 服务，来重试整个调度过程。

整个调度过程可以分为 3 个主要阶段：预调度阶段主要进行安全检查，并为将要进行的调度过程准备相应的数据项和请求对象；在调度阶段 Nova 可能会进行超过一次的调度决策，最终将确定系统是否有能力创建相应的虚拟机，以及该创建在哪个主机上；当调度决策完成后，nova-compute 会在选择的主机上真正消耗资源，启动虚拟机和对应的网络存储设备。在冷迁移、热迁移、Resize、Rebuild 和 Evacuate 过程中，各个子服务的功能和职责都是类似的，详见第 5.5 节的内容。

关于整个调度子系统在超大规模 OpenStack 环境中的性能分析和优化情况，可参考文章 http://01.org/sites/default/files/performance_analysis_and_tuning_in_china_mobiles_openstack_production_cloud.pdf。

5.5 典型工作流程

5.5.1 创建虚拟机

创建一个虚拟机至少需要指定的参数有 3 个：虚拟机名字、镜像和 Flavor。执行“openstack image list”命令可以看到目前可用的虚拟机镜像，如图 5-6 所示。

```
stack@bianst:/opt/stack/nova$ nova image-list
```

ID	Name	Status	Server
1de090e2-3cab-4ca3-82ce-7167e61c1bd4	Fedora-x86_64-20-20140618-sda	active	
394c3019-d602-4096-a2d5-71ebfe8a5c75	cirros-0.3.2-x86_64-uec	active	
86872e30-1bac-41dc-afce-683dfbb471d5	cirros-0.3.2-x86_64-uec-kernel	active	
225ed79b-0a35-4ace-956a-92de69b3f6bd	cirros-0.3.2-x86_64-uec-ramdisk	active	

图 5-6 虚拟机镜像列表

比如创建一个名为 test 的虚拟机，使用 Flavor 类型 ml.tiny:

```
$ nova boot --flavor ml.tiny --image cirros-0.3.4-x86_64-uec test
```

创建之后可以执行“nova list”查看虚拟机是否运行正常，如图 5-7 所示。

ID	Name	Status	Task State	Power State	Networks
2dd52014-78f5-4d9b-a65e-8190421d89d6	test	ACTIVE	-	Running	private=10.0.0.2

图 5-7 nova list

在 KVM 环境下，使用 Libvirt 库提供的 virsh 命令创建一个虚拟机需要一个 XML 配置文件。在 Nova 中创建一个虚拟机同样也需要这个 XML 配置文件，不同的是这个 XML 配置文件由 Nova 根据用户设定的参数自动生成。

Nova 生成的 XML 配置文件都存放在/opt/stack/data/nova/instances/目录，这些配置文件很好地体现了 Nova 在创建虚拟机时需要哪些参数信息。

```
/opt/stack/data/nova/instances/e16fc90a-31a0-44cc-a48b-5a33ddf99ed1$ ls
console.log disk disk.config kernel libvirt.xml ramdisk
```

“e16fc90a-31a0-44cc-a48b-5a33ddf99ed1”表示新建虚拟机的 UUID，这个目录下存放了该虚拟机的内核镜像等文件。

虚拟机创建过程如图 5-8 所示。

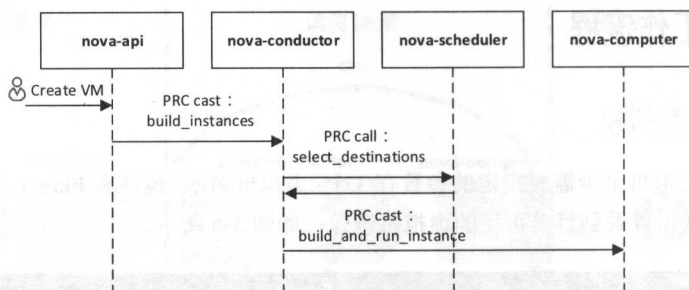


图 5-8 虚拟机创建过程

如前所述，创建虚拟机等 TaskAPI 任务，已经由 nova-conductor 承担，因此 nova-api 监听到创建虚拟机的 HTTP 请求后，会通过 RPC 调用 nova.conductor.manager.ComputeTaskManager 中的 build_instances()方法。

nova-conductor 会在 build_instances()中生成 request_spec 对象，其中包括详细的虚拟机信息，nova-scheduler 依据这些信息为虚拟机选择一个最佳的主机，然后 nova-conductor 再通过 RPC 调用 nova-compute 创建虚拟机。

nova-compute 首先会使用 Resource Tracker 的 Claim 机制检测一下主机的可用资源是否能够满足新建虚拟机的需要，然后通过具体的 Virt Driver 创建虚拟机。

5.5.2 冷迁移与 Resize

迁移是指将虚拟机从一个计算节点迁移到另外一个节点上。冷迁移是相对于热迁移而言，区别在于在冷迁移过程中虚拟机是关机或是处于不可用的状态，而热迁移则需要保证虚拟机时刻运行。

Resize 则是指根据需求调整虚拟机的计算能力和资源。Resize 和冷迁移的工作流程相同，区别只是 Resize 时必须保持新的 Flavor 配置大于旧的配置，而冷迁移则要求两者相同。Resize 的工作流程如图 5-9 所示。

nova-api 将虚拟机的状态修改为 RESIZE_PREP，Resize 与冷迁移属于 TaskAPI 任务，因此 nova-api 会通过 nova.conductor.rpcapi.ComputeTaskAPI 提供的 RPC 接口 migrate_server()调用 nova-conductor。

nova-conductor 根据参数选择 Resize 的流程，生成 request_spec 对象，并调用 nova-scheduler 选择一个合适的目标主机，最后调用目标主机的 nova-compute。

目标主机上要做一些准备的动作，比如通过 Resource Tracker 的 Claim 机制检测一下主机是否满足条件等，之后通过 RPC 回到源主机上由源主机的 nova-compute 服务完成迁移。

在源主机上，nova-compute 获取虚拟机的磁盘、网络等信息，使用 cp 或 scp 命令复制需要迁移的资源到目标主机，修改虚拟机的状态为 RESIZE_MIGRATED，再次通过 RPC 到目标主机上完成虚拟机的 Resize。

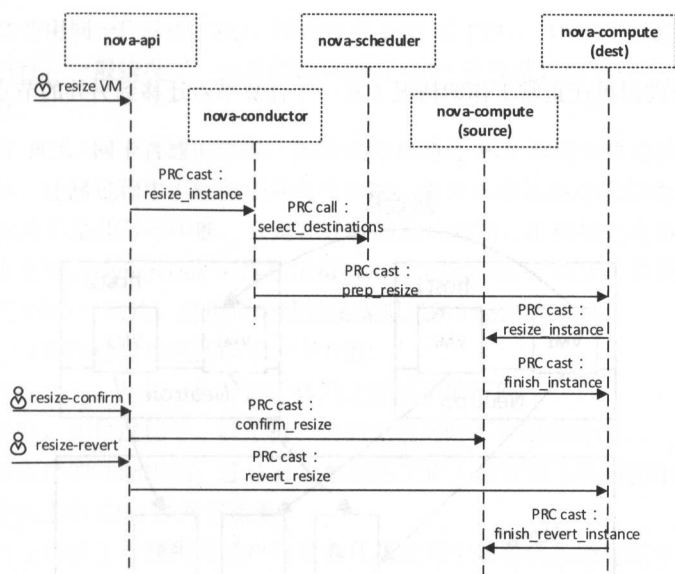


图 5-9 Resize 工作流程

目标主机的 nova-compute 根据新虚拟机的参数信息准备资源并创建，然后将虚拟机的状态修改为 RESIZED。

管理员需要确认是否完成 Resize，可以有两种选择，确认完成或回退。当确认时会清理源主机上的资源。如果选择回退，则首先到目标主机上清理资源，再到源主机上恢复到未 Resize 时的状态。

如果希望能够在本地 Resize，则必须在 /etc/nova/nova.conf 中配置 “allow_resize_to_same_host” 选项的值为 “True”，默认情况下，执行的 Resize 都是针对非本地，即将虚拟机从一个源主机迁移到另一个目标主机的情况。执行 Resize 时：

```

/opt/stack/data/nova/instances$ ls
456d21b4-1707-4a08-b4f0-4d36df3cb84b _base compute_nodes
locks 456d21b4-1707-4a08-b4f0-4d36df3cb84b _resize
  
```

源主机上的虚拟机 “456d21b4-1707-4a08-b4f0-4d36df3cb84b” 会首先被关闭，并复制一份为 “456d21b4-1707-4a08-b4f0-4d36df3cb84b_resize”，然后以新的副本为基础完成到目标主机的复制。

如果源主机和目标主机共享存储，则源主机会直接在共享的存储上使用 mkdir 命令建立新目录 “456d21b4-1707-4a08-b4f0-4d36df3cb84b”，否则，源主机需要通过 SSH 连接到目标主机建立这个目录。

新目录成功建立后，源主机需要将虚拟机的镜像转换为 RAW 格式，并使用 cp（共享存储时）或 scp 命令将其复制到刚才新建的目录里。

5.5.3 热迁移

热迁移是指虚拟机在正常工作的情况下从一个计算节点迁移到另外的节点上，如图 5-10 所示。

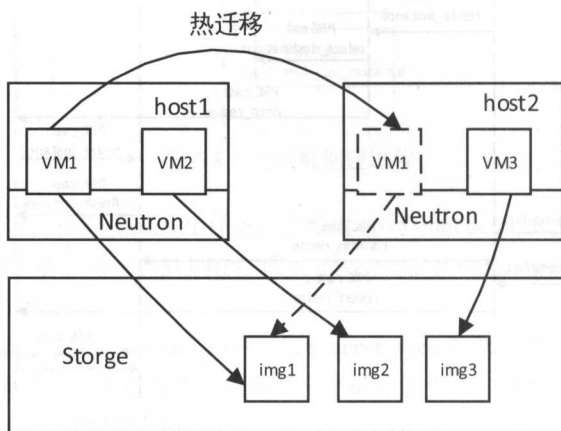


图 5-10 热迁移示意图

host1 与 host2 共享存储 Storage，虚拟机 VM1 从 host1 主机迁移到 host2 主机，在业务不中断的条件下将虚拟机的内存复制到 host2 主机上，镜像的存储路径并没有发生改变。

虚拟机的热迁移在生产环境中有着极大的用处，在各个厂商的产品中，热迁移已经成为一个重要指标。热迁移有很多的特点：

- 动态调整每个计算节点的负载，使资源得到最大限度地使用。例如通过检测虚拟机 CPU 使用的情况，把空闲的虚拟机迁移到一些节点上，这样可以关闭一些节点，在这个资源匮乏的时代，节能就是利润。
- 在线升级以及节点维护。很多云系统和 OpenStack 一样，升级时都需要重启服务，有的甚至需要重启节点，但是有些应用场景，比如说移动的服务器是要保证业务的几乎 100% 不中断的，即使在深夜也是不允许业务中断，要升级怎么办，如果没有热迁移特性，需要不同机房之间做业务备份，耗时耗材。使用热迁移特性，在深夜业务量小的时候，将一部分节点的虚拟机迁移到其他节点上，升级完之后再迁移过来，然后再升级另外一部分。

由于热迁移要求虚拟机业务不中断，所以一般都是在共享存储的条件下，这时影响热迁移的关键因素有两个：一是虚拟机内存脏页的速度，迭代复制是以页为单位；二是网络带宽。如果脏页的速度远大于迭代复制内存页的速度，在一段时间内迁移是不成功的。

两个计算节点之间的虚拟机是否可以自由迁移，还有其他因素的制约。

- CPU 兼容性：不同厂家，不同系列的 CPU 兼容性需要保证，一般生产环境的一个集

群都是使用同一厂家的 CPU，即使都是 Intel 的 CPU，不同型号之间的 CPU 也有不同的特性，一般情况下，如果源计算节点 CPU 特性是目标节点 CPU 特性的子集即可迁移。

- 是否有 PCI，网卡直通的情况：如果虚拟机通过 PCI 和网卡直通技术直接使用物理的设备，迁移过程中不能保证业务不中断，也无法满足热迁移的要求。

其实热迁移并不是业务不中断，只是在迁移的最后时刻，虚拟机会短暂挂起，快速完成最后一次内存复制。Hypervisor 中挂起虚拟机本质上就是改变 VCPU 的调度，暂时不给虚拟机可用的物理 CPU 时间片。给用户的感觉是虚拟机瞬间无响应。

虚拟机热迁移的性能指标包括以下 3 个方面。

- 整体迁移时间：从源主机开始迁移到迁移结束的时间。
- 停机时间：迁移过程中，源主机、目的主机同时不可用的时间。
- 对应用程序的性能影响：迁移对于被迁移主机上运行服务性能的影响程度，数据复制会冲高主机 CPU 和网络流量。

热迁移和冷迁移的工作流程类似，只是热迁移过程中兼容性判断比较多，而冷迁移其实就是使用原来所需的资源在目标节点上重新创建一个虚拟机，这样很多兼容性问题是不需要考虑的。热迁移工作流程如图 5-11 所示。

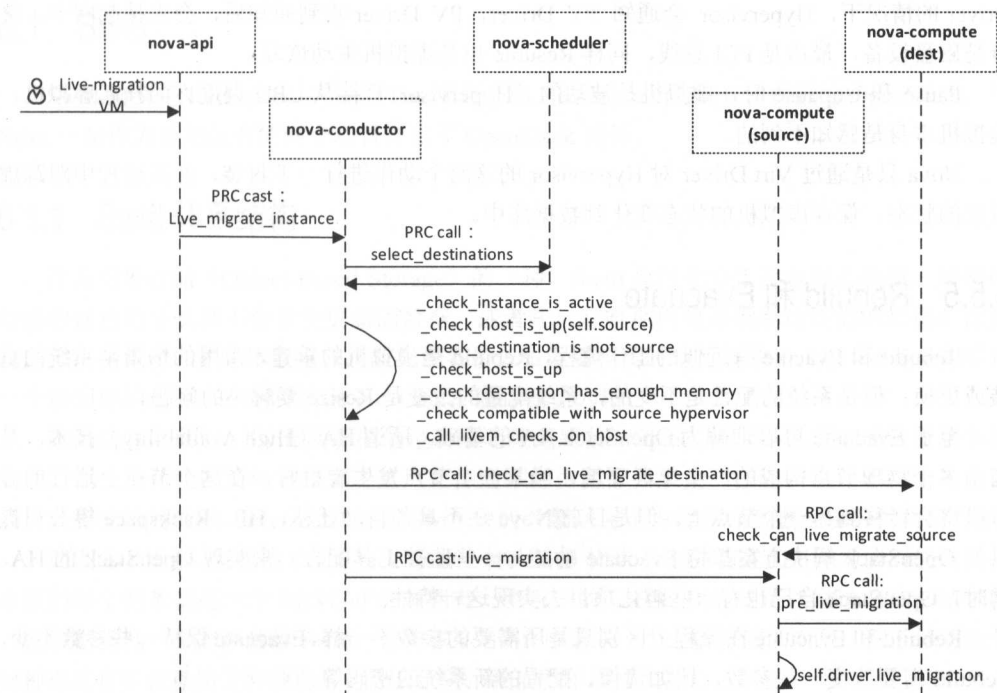


图 5-11 热迁移工作流程

与虚拟机创建、冷迁移、Resize 一样，热迁移也属于 TaskAPI 任务，nova-api 将虚拟机的状态为 MIGRATING 后，通过 nova.conductor.rpcapi.ComputeTaskAPI 提供的 RPC 接口 migrate_server()调用 nova-conductor。

nova-conductor，首先检查虚拟机的状态（必须是正常运行 RUNNING），如果没有指定源主机，和冷迁移一样调用 nova-scheduler 选择一个可用的主机，然后调用 nova-compute 完成迁移。

在完成迁移之前，nova-compute 还有很多工作，比如在源主机执行动作之前还需要到目标主机上验证是否满足迁移的条件，判断磁盘是否是共享、是否是 block 迁移等，目标主机将会验证的结果返回给源主机。然后源主机 nova-compute 调用 Virt Driver 比如 Libvirt 的接口完成最终的迁移动作。

5.5.4 挂起和恢复

从 Hypervisor 的角度有两种挂起和恢复虚拟机的方式，挂起是 Suspend 和 Pause，对应的恢复是 Resume 和 Unpause。

Suspend 和 Pause 都是挂起虚拟机，从 Hypervisor 的角度看，两者的区别是 Suspend 会通知到虚拟机，虚拟机内部进行挂起的动作，比如 Xen 环境下 Window 7 的虚拟机，在安装 PV Driver 的情况下，Hypervisor 会通知 PV Driver，PV Driver 收到通知后，会先挂起网卡，接着是磁盘设备，最后是 PCI 总线，同样 Resume 也是虚拟机主动恢复。

Pause 和 Unpause 时，虚拟机是被动的，Hypervisor 直接从 CPU 调度方面挂起虚拟机，虚拟机本身是感知不到的。

Nova 只是通过 Virt Driver 对 Hypervisor 的这两个动作进行一下封装，并在过程中跟踪虚拟机的状态，保存虚拟机的状态变化到数据库中。

5.5.5 Rebuild 和 Evacuate

Rebuild 和 Evacuate 有近似的工作流程，Rebuild 是虚拟机的重建，常用的场景是系统的重装或更换，但是系统的配置是不变的，系统配置的改变是 Resize 要解决的问题。

至于 Evacuate 可以理解为 OpenStack HA 的基础。所谓 HA（High Availability）技术，是指由多个物理节点构成的一个集群环境，当某一个节点发生宕机时，在这个节点上运行的虚拟机将会转移到另一个节点上。但是目前 Nova 还不具备自动迁移，HP、Rackspace 等公司提供的 OpenStack 解决方案都将 Evacuate 功能与一些监控工具配合，来实现 OpenStack 的 HA。同时，OpenStack 自己也有一些孵化项目去实现这一特性。

Rebuild 和 Evacuate 在流程上区别只是所需要的参数不一样，Evacuate 保持一些参数不变，Rebuild 需要改变一些参数，比如镜像、配置的新系统的密码等。

存储是 OpenStack 所管理的最重要的资源之一。

Nova 实现了 OpenStack 虚拟机世界的抽象，并利用主机的本地存储为虚拟机提供“临时存储（Ephemeral Storage）”。如果虚拟机被删除了，挂在这个虚拟机上的任何临时存储都将自动释放。存放在临时存储上的数据是高度不可靠的，任何虚拟机和主机的故障都可能会导致数据丢失。因此，基于临时存储的虚拟机就是无根之浮萍，没有确切的归属，在它生命周期终止的时候，所有发生在它身上的故事以及一切的痕迹都将被抹去。

而基于 SAN、NAS 等不同类型的存储设备，Swift（对象存储）与块存储（Cinder）引入了“永久存储（Persistent Storage）”，共同为这个虚拟机世界的主体——虚拟机提供了安身之本，负责为每个虚拟机本身的镜像以及它所产生的各种数据提供一个家，尽量地去做做到“居者有其屋”。

6.1 Swift

Swift 前身是 Rackspace Cloud Files 项目，由 Rackspace 于 2010 年贡献给 OpenStack，与 Nova 一起作为最初仅有的两个项目开启了 OpenStack 元年。

6.1.1 Swift 体系结构

作为对象存储（Object-Based Storage）的一种，Swift 比较适合于存放静态数据。所谓的静态数据指的是长期不会发生更新的数据，或者在一定时期内更新频率比较低的数据。比如说虚拟机的镜像、多媒体数据以及数据的备份。如果需要实时地更新数据，那么 Swift 并不是一个特别好的选择。在这种情况下，Cinder 块存储更为适合。

既然是对象存储，Swift 所存储的逻辑单元就是对象（Object），而不是我们通常概念中的文件。在一个传统的文件系统实现里，文件通常都由两部分来共同描述：文件本身的内容以及与其相关的元数据（Metadata）。而 Swift 中的对象涵盖了内容与元数据两部分的内容。

如图 6-1 所示，与其他 OpenStack 项目一样，Swift 提供了 RESTful API 作为访问的入口，存储的每个对象都是一个 RESTful 资源拥有一个唯一的 URL，我们既可以发送 HTTP 请求将一些数据作为一个对象传给 Swift，也可以从 Swift 中请求一个之前存储的对象。至于该对象是何种形式存在且存储于何种设备的什么位置，我们并不需要去关心。

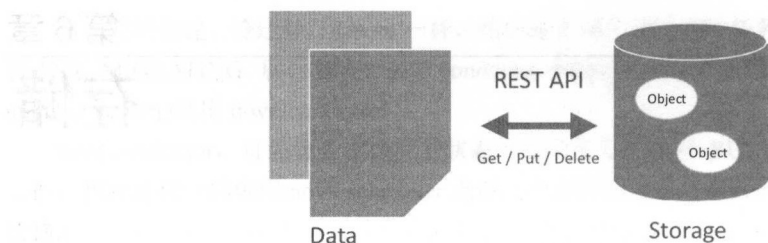


图 6-1 对象存储

如图 6-2 所示, Swift 从架构上可以划分为两个层次: 访问层(Access Tier)与存储层(Storage Nodes)。访问层的功能类似于网络设备中的 Hub, 主要包括两个部分, 即 Proxy Node (代理服务节点) 与 Authentication (认证), 分别负责 RESTful 请求的处理与用户身份的认证。

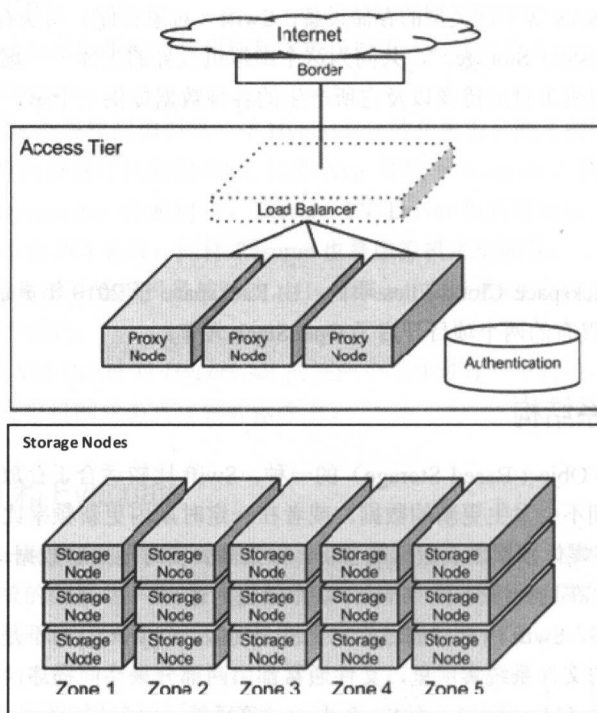


图 6-2 Swift 架构

Proxy Node 上运行有 Proxy Server 来负责处理用户的 RESTful 请求, 接收到用户请求时, 需要对用户的身份进行认证, 此时用户所提供的身份资料会被转发给认证服务进行处理。Proxy Server 可以使用 Memcached (高性能的分布式内存对象缓存系统) 进行数据和对象的缓

存, 来减少数据库读取的次数, 提高用户的访问速度。

存储层由一系列的物理存储节点组成, 负责对象数据的存储。Proxy Node 收到用户的访问请求时, 会将其转发到相应的存储节点上。为了在系统出现问题的情况下有效地将故障隔离在最小的物理范围内, 存储层在物理上又分为如下一些层次。

- **Region:** 地理上隔绝的区域, 也就是说不同的 Region 通常在地理位置上被隔绝开来。比如说两个数据中心可以划分为两个 Region。每个 Swift 系统默认至少有一个 Region。
- **Zone:** 在每个 Region 的内部又划分了不同的 Zone 来实现硬件上的隔绝。一个 Zone 可以是一个硬盘、一台主机、一个机柜或者是一个交换机, 但是我们可以简单理解为一个 Zone 代表了一组独立的存储节点。
- **Storage Node:** 存储对象数据的物理节点, 基于通用标准的硬件设备提供了不亚于专业存储设备的对象存储服务。
- **Device:** 可以简单理解为磁盘。
- **Partition:** 这里的 Partition 仅仅是指在 Device 上的文件系统上的目录, 和我们通常所理解的硬盘分区是完全不同的概念。

如图 6-3 所示, 每个 Storage Node 上存储的对象在逻辑上又由 3 个层次组成: Account、Container 和 Object。

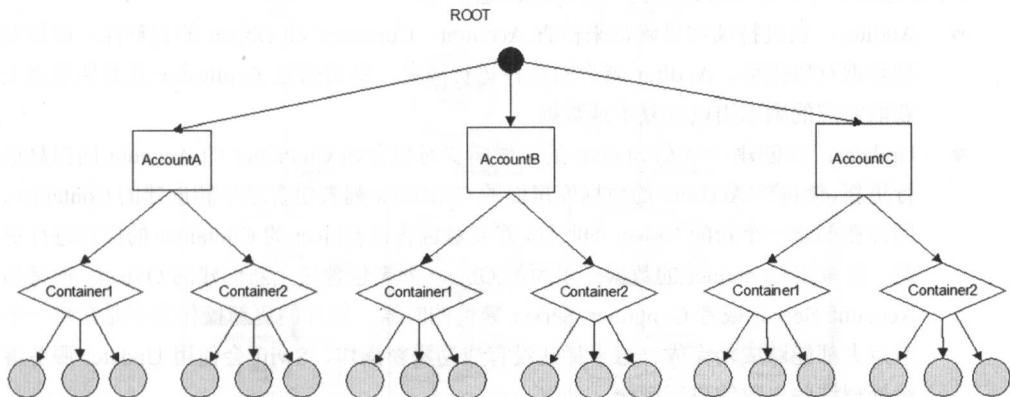


图 6-3 Swift 对象组织结构

这里的每一层所包含的节点数都没有限制, 可以任意进行扩展。Account 在对象的存储过程中实现顶层的隔离, 所代表的并不是个人账户, 而是租户。一个 Account 可以被多个个人账户所共同使用; Container 代表了一组对象的封装, 类似文件夹或目录, 但是 Container 不能嵌套, 并不能包含下级的 Container; 位于最后一个层次的即是具体的对象, 由元数据和内容两部分组成。Swift 要求一个对象必须存储在某个 Container 中, 因此一个 Account 应该有至少一个 Container 来提供对象的存储。

与上述的 3 层组织结构相对应, Storage Node 上运行有 3 种服务。

- **Account Server:** 提供 Account 相关服务, 包括所含 Container 列表以及 Account 的元数据等。Account 的信息被存储在一个 SQLite 数据库中。
- **Container Server:** 提供 Container 相关服务, 包括所含 Object 的列表以及 Container 的元数据等。与 Account 一样, Container 的信息也被存储在一个 SQLite 数据库中。
- **Object Server:** 提供对象的存取和元数据服务, 每个对象的内容会以二进制文件的形式存储在文件系统中, 元数据会作为文件的扩展属性来存储, 也就是说, 存储对象的物理节点上, 本地文件系统必须支持文件的扩展属性, 有些文件系统 (比如 Ext3) 的文件扩展属性默认是关掉的。

为了保证数据在某个存储硬件损坏的情况下也不会丢失, Swift 为每个对象都建立了一定数量的副本 (Replica, 默认为 3 个), 并且每个副本存放在不同的 Zone 中, 这样即使某个 Zone 发生故障, Swift 也仍然可以通过其他 Zone 继续提供服务。

在 Swift 中, 副本是以 Partition 为单位的。也就是说, 对象的副本其实是通过 Partition 的副本来实现的。Swift 管理副本的粒度是 Partition, 而非单个对象。

既然一个对象并不是只保存了一份, 那么对象和其副本之间的数据一致性问题就必须得到解决, 对象内容更新的时候副本也必须同时更新, 而且其中一个损坏时必须能迅速复制一份来完整替换。

Swift 通过 3 种服务来解决数据一致性的问题。

- **Auditor:** 通过持续扫描磁盘来检查 Account、Container 和 Object 的完整性。如果发现数据有所损坏, Auditor 就会对文件进行隔离, 然后通过 Replicator 从其他节点上获取对应的副本用以恢复本地数据。
- **Updater:** 在创建一个 Container 的时候需要对包含该 Container 的 Account 的信息进行更新, 使得该 Account 的数据库里面的 Container 列表包含这个新创建的 Container。同样在创建一个新的 Object 的时候, 需要对包含该 Object 的 Container 的信息进行更新, 使得该 Container 的数据库里面的 Object 列表包含这个新创建的 Object。但是当 Account Server 或者 Container Server 繁忙的时候, 这样的更新操作并不是在每一个节点上都能够成功完成。对于那些没有成功更新操作, Swift 会使用 Updater 服务继续处理这些失败的更新操作。
- **Replicator:** 负责检测各个节点上的数据及其副本是否一致。当发现不一致时会将过时的副本更新为最新版本, 并且负责将标记为删除的数据真正从物理介质上删除。

目前为止, 我们可以看到有 Proxy Server 处理用户的对象存取请求, 有认证服务负责对用户的身份进行认证, Proxy Server 接收到用户请求后, 又会把请求转发给存储节点上的 Account Server、Container Server 与 Object Server 进行具体的对象操作, 而对象与其各个副本之间数据的一致性则由 Auditor、Updater 与 Replicator 来负责。

虽然对象最终仍然以文件的形式存储在存储节点上, 但是 Swift 中并没有“路径”以及“文件夹”这样传统文件系统中的概念, 那么剩下的问题就是, Swift 如何将对象与真正的物理存

储位置相映射？Swift 引入了环（Ring）的概念来解决这个问题。

Ring 记录了存储对象与物理位置之间的映射关系，Account、Container 和 Object 都有自己独立的 Ring。当 Proxy Server 接收到用户请求时，会根据所操作的实体（Account、Container 或 Object）寻找对应的 Ring，来确定它们在存储服务器集群中的具体位置。至于 Account、Container 和 Object 在 Ring 中的位置信息，也由 Proxy Server 来进行维护。

Ring 通过 Zone、Device、Partition 和 Replica 的概念来维护映射信息。每个 Partition 的位置由 Ring 来维护，并存储在映射中。Ring 需要在 Swift 部署时使用“swift-ring-builder”工具手动构建，之后每次增减存储节点时，都需要重新平衡（Rebalance）一下 Ring 文件中的项目，以保证系统因此而发生迁移的文件数量最少。

现在，图 6-2 所示的 Swift 架构可以演化为图 6-4 所示的形式。

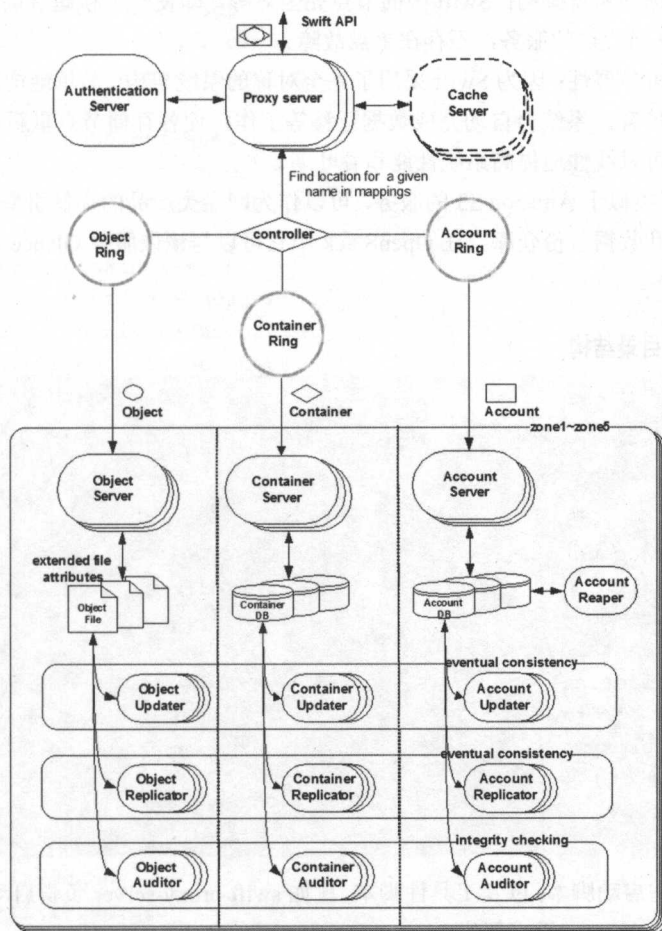


图 6-4 Swift 架构

Proxy Server 是运行在 Proxy Node 上的 WSGI Server, Account Server、Container Server 与 Object Server 是运行在存储节点上的 WSGI Server, Proxy Server 收到用户 HTTP 请求后, 将请求路由到相应的 Controller (AccountController、ContainerController 与 ObjectController), Controller 会从对应的 Ring 文件中获取到请求数据所在的存储节点, 然后将这个请求转发给该节点上的 WSGI Server (Account Server、Container Server 与 Object Server)。

由上述的 Swift 架构可以看出, Swift 的设计有很多的优点。

- 极高的数据持久性: 数据的持久性是指数据存储到系统中后, 到某一天数据丢失的可能性。比如 Amazon S3 的数据持久性是 11 个 9, 即如果存储 1 万个 (4 个 0) 文件到 S3 中, 1 千万 (7 个 0) 年之后, 可能会丢失其中 1 个文件。Swift 通过采用副本等冗余技术达到了极高的数据持久性。
- 完全对称的系统架构: Swift 中的节点完全对等, 即使一个存储节点宕机, 也不会影响 Swift 所提供的服务, 不存在单点故障。
- 无限的可扩展性: 因为 Swift 采用了完全对称的系统架构, 简单地增加节点即可实现系统的扩容, 系统会自动完成数据迁移等工作, 使各存储节点重新达到平衡状态, 同时也可以线性地提高系统性能和吞吐率。

Swift 提供了类似于 Amazon S3 的服务, 可以作为网盘类产品的存储引擎, 也非常适合用于存储日志文件和数据备份仓库。在 OpenStack 中它可以与镜像服务 Glance 相结合, 为其存储镜像文件。

1. Swift 源码目录结构

```
.
├── api-ref
├── bin
├── etc
├── swift
│   ├── account
│   ├── cli
│   ├── common
│   ├── container
│   ├── locale
│   ├── obj
│   └── proxy
├── setup.cfg
├── setup.py
└── test
```

- bin: 一些启动脚本, 以及工具性脚本。比如 swift-proxy-server 负责启动 proxy server, swift-ring-builder 用来创建 Ring。
- swift: swift 的核心代码。account、container、obj、proxy 几个子目录分别是 Account、

Container、Object、Proxy 几个服务的具体实现。common 子目录包含的是一些可以被多个组件共用的公共代码，比如说 Account、Container 以及 Object 服务都有 Ring 的操作，那么这些共用的代码就存放在 common 目录下。

- etc: 配置文件模板，包括 Paste 配置文件等。

2. setup.cfg

依照惯例，理解具体的实现之前，我们需要仔细浏览 setup.cfg 文件。

```
scripts =
    bin/swift-account-audit
    bin/swift-account-auditor
    bin/swift-account-info
    bin/swift-account-reaper
    bin/swift-account-replicator
    bin/swift-account-server
    bin/swift-config
    bin/swift-container-auditor
    bin/swift-container-info
    bin/swift-container-replicator
    bin/swift-container-server
    bin/swift-container-sync
    bin/swift-container-updater
    bin/swift-container-reconciler
    bin/swift-reconciler-enqueue
    bin/swift-dispersion-populate
    bin/swift-dispersion-report
    bin/swift-drive-audit
    bin/swift-form-signature
    bin/swift-get-nodes
    bin/swift-init
    bin/swift-object-auditor
    bin/swift-object-expirer
    bin/swift-object-info
    bin/swift-object-replicator
    bin/swift-object-server
    bin/swift-object-updater
    bin/swift-oldies
    bin/swift-orphans
    bin/swift-proxy-server
    bin/swift-recon
    bin/swift-recon-cron
    bin/swift-ring-builder
    bin/swift-ring-builder-analyzer
```

对于 Swift 来说, `setup.cfg` 文件中值得关注的是 `scripts` 关键字所对应的内容, 其中的每一项都代表了一个被安装在系统里的可执行脚本。与前面所介绍的架构对照分析, 我们可以更好地理解 Swift 的工作, 它们同时也是 Swift 各项工作的入口, 完全可以作为我们理解 Swift 具体实现的起点。

- `swift-proxy-server`: 代理服务 (Proxy Server) 进程, 通常运行这个进程的服务器又被称为代理服务器。
- `*-auditor`: `swift-account-auditor`、`swift-container-auditor`、`swift-object-auditor` 分别对应了 Account、Container 和 Object 的 Auditor (审计) 进程。
- `*-updater`: `swift-container-updater` 与 `swift-object-updater` 分别对应了 Container 和 Object 的 Updater 进程, 并不存在 Account 的 Updater。
- `*-replicator`: 基于 Account、Container 和 Object 存储方式的不同, 它们的 Replicator 进程又分为两类, 一是针对 Account 与 Container 这两种以数据库形式存在的数据, `swift-account-replicator` 与 `swift-container-replicator` 完成的是数据库的复制; 二是 `swift-object-replicator` 针对 Object 完成对象数据的复制。
- `swift-account-server`、`swift-container-server` 与 `swift-object-server`, 分别对应了 Account、Container 和 Object 3 种服务进程。
- `swift-account-audit`: 与 `*-auditor` 不同, `swift-account-audit` 并不是一个后台服务进程, 它只是一个命令行工具, 可以用来手动地对指定的 Account、Container 或 Object 的数据进行完整性检测。指定了 Container 时, 将递归检测该 Container 中的每个 Object, 同理, 指定了 Account 时, 将会递归检测该 Account 下的每个 Container, 并进一步递归检测每个 Container 中的各个 Object。
- `*-info`: 打印 Account、Container 和 Object 的内容或元数据等信息。
- `swift-account-reaper`: Account 收割器, 负责清除被删除 Account 中所包含的数据。
- `swift-container-sync`: 实现 Container 同步功能。一个 Container 的所有内容可以通过后台同步镜像到另一个 Container (两个 Container 可以在同一个集群也可以在完全不同的集群)。比如可以使用这个功能实现 Account 的迁移, 将旧 Account 中的所有 Container 都同步到新的 Account 中。利用这个功能, 数据也可以在不同的云服务提供商之间迁移, 而不会被锁定在一个特定的供应商, 比如数据可以从私人的 Swift 集群迁移到一个公共 Swift 云。
- `swift-container-reconciler`: 伴随 Storage Policies (存储策略) 而出现的一个后台进程, 简单地说, 一种 Storage Policy 就是一种存储的方式, 比如创建两个副本或创建 3 个副本, `swift-container-reconciler` 主要负责将位于错误 Storage Policy 中的对象迁移到正确的 Storage Policy 中。
- `swift-dispersion-report`: 通过检测 Container 和 Object 是不是在集群中合适的位置来估计整

个集群的健康状态。比如说，每个对象有 3 个副本，如果 3 个副本中只有两个在适当的位置上，那么这个对象的健康值就是 66.66%，最佳的是 100%。如果我们在占集群一定百分比的 Partition 中（比如 1%）创建足够的对象，就可以获得一个对整个集群健康度的相当有效的估计。为了估计健康状态，我们所要做的第一件事就是创建一个专门用于该工作的 Account，然后，我们需要向不同的 Partition 中放置容器和对象。swift-dispersion-populate 工具能完成这个工作，它创建随机的 Container 和 Object 直到它们落在不同的 Partition 里。最后，我们运行 swift-dispersiton-tool 来进行检测。

- swift-drive-audit: 经验表明，当一个设备快要出故障时，错误信息会涌入/var/log/kern.log。此时我们可以通过 cron 运行 swift-drive-audit 脚本来寻找坏掉的硬盘，并将其卸载，从而 Swift 能够绕开它工作。
- swift-get-nodes: 如果我们希望知道对象在哪个存储节点上，可以使用这个工具。
- swift-init: 启动 swift 的基本服务。
- swift-object-expirer: 在指定时间内删除对象。当对象过期达到删除的时间节点时，Swift 会停止向用户提供针对该对象的服务，并在短时间内把该对象删除。
- swift-oldies: 可以列出运行时间很长的 Swift 服务进程，比如“swift-oldies -a 48”打印出已经运行超过 48 个小时的那些进程。
- swift-orphans: 可以用于清除那些孤儿进程（父进程已经退出）。
- swift-recon: 可以用于收集一些 Swift 统计数据，比如 Account、Container 和 Object 的数量等。
- swift-ring-builder: 用于构建 Ring。
- swift-ring-builder-analyzer: 用于快速地测试不同的 Ring 配置。

6.1.2 Ring

Swift 通过引入 Ring 来实现对物理节点的管理，包括记录对象与物理存储位置间的映射，物理节点的添加和删除等。

针对决定某个对象存储在哪个节点上之类的问题，最常规的做法就是采用 Hash 算法。如果存储节点的数量固定，普通的 Hash 算法就能满足要求，但是因为 Swift 通过增减存储节点来实现无限的可扩展性，存储节点的数量可能会发生变动，此时所有对象的 Hash 值都会改变，这对于部署了极多节点的 Swift 来说不太现实。因此 Swift 采用了一致性 Hash 算法来构建 Ring。

1. 一致性 Hash

假设有 N 台存储节点，为使负载均衡，需要把对象均匀地映射到每个节点上，这通常使用 Hash 算法来实现：对于普通的 Hash 算法，首先计算对象的 Hash 值 Key，然后再计算 $\text{Key} \bmod N$ （Key 对 N 取模）的结果，得到的余数即为数据存放的节点。比如， N 等于 2，则值为 0、1、2、3、4 的 Key 按照取模的结果分别将存放在 0、1、0、1、0 号节点上。如果哈希算

法是均匀的，数据就会平均分配在两个节点中。如果每个数据的访问量比较平均，负载也会平均分配到两个节点上。

当然，这只是理想中的情况。在实际使用中，当数据量和访问量进一步增加，两个节点无法满足需求的时候，需要增加一个节点来服务用户的访问请求，此时 N 增加为 3，映射关系变为 $\text{Key} \bmod (N+1)$ ，上述的哈希值为 2、3、4 的对象就需要重新分配。如果存储节点的数量，以及对象的数量很多时，迁移所带来的代价会非常大。

为了减少节点增减所带来的代价，Swift 采用了一致性 Hash 算法，在存储节点的数量发生改变时，尽量少地改变已经存在的对象与节点间的映射关系，从而大大减少需迁移的对象数量。

一致性 Hash 的过程由如下几个步骤组成：

- 计算每个对象名称的 Hash 值并将它们均匀地分布到一个虚拟空间上去，一般用 2 的 32 次幂标识该虚拟空间。
- 假设有 2^m 个存储节点，那么将虚拟空间均匀分成 2^m 等份，每一份长度为 $2^{(32-m)}$ 。
- 假设一个对象名称 Hash 之后的结果是 n ，那么该对象对应的存储节点即为 $n/2^{(32-m)}$ ，转换为二进制位移操作，就是将 Hash 之后的结果向右位移 $(32-m)$ 位。

图 6-5 以 $m=3$ 为例，演示了具体的映射过程。一般将虚拟空间用一个环（Ring）表示，这也是为什么 Swift 用 Ring 来表示从对象到物理存储位置的映射。

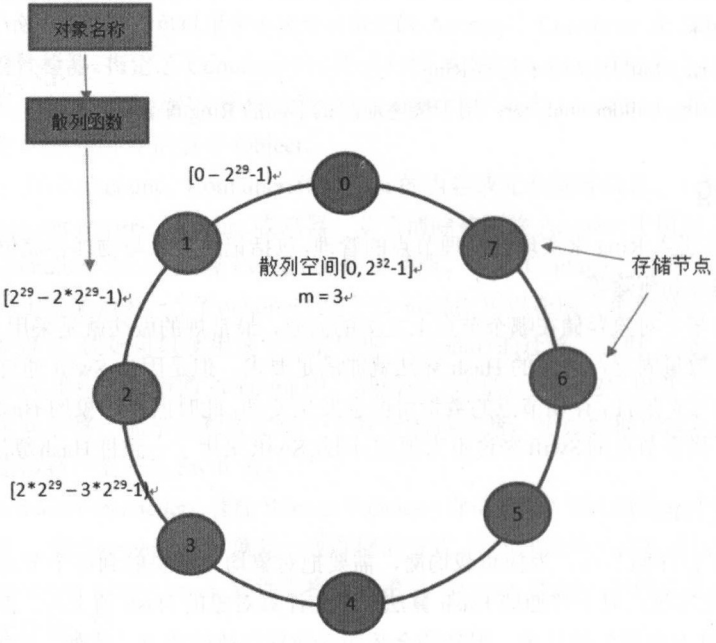


图 6-5 一致性 Hash

2. Ring 数据结构

Swift 中所谓的 Ring 就是基于一致性 Hash 所构造的环，如图 6-6 所示。Ring 包括以下 3 种重要的数据结构（信息）。

- 设备表：Swift 将所有 Device 编号，设备表中的每一项都对应一个 Device，其中记录了该 Device 的具体位置信息，包括 Device ID、所在的 Region、Zone、IP 地址以及端口号，以及用户为该 Device 定义的权重（Weight）等。
- Device 的容量大小不一时，可以通过 Weight 值保证 Partition 均匀分布，容量较大的 Device 拥有更大的权重，也容纳更多的 Partition。比如，一个 1TB 大小的 Device 有 100 的权重，而一个 2TB 大小的 Device 将有 200 的权重。
- 设备查询表（Device Lookup Table）：存储 Partition 的各个副本（默认为 3 个）与具体 Device 的映射信息。设备查询表中的每一列对应一个 Partition，每一行对应 Partition 的一个副本，每个表格中的信息设备表中 Device 的编号，根据这个编号，可以去设备表中检索到该 Device 的具体连接信息（Device ID，IP 以及端口号等信息）。
- Partition 移位值（Partition Shift Value）：表示在 Hash 之后将 Object 名字二进制移位的位数。

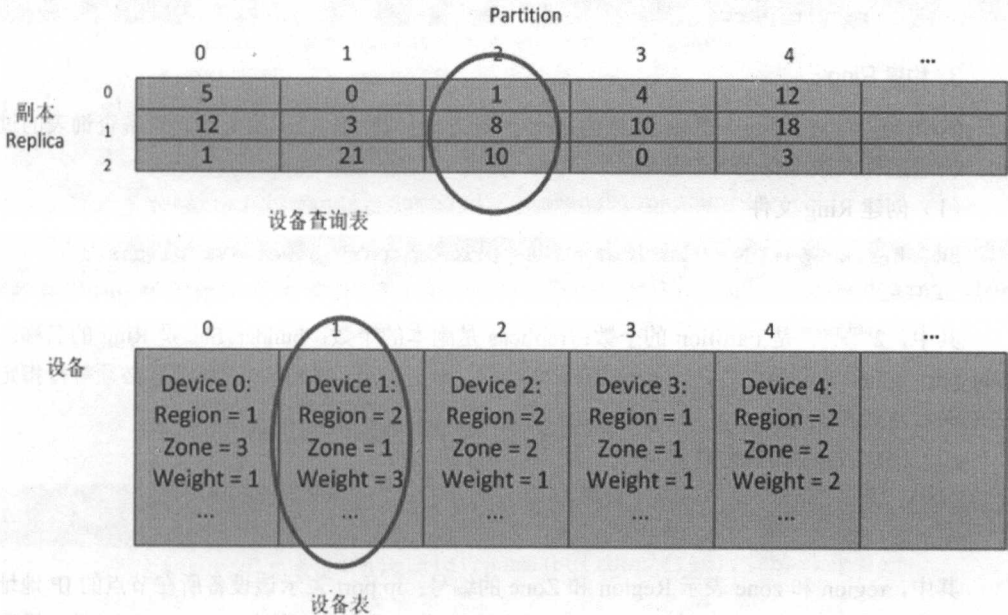


图 6-6 Ring 数据结构

为了减少由于增加、减少节点所带来的数据迁移，Swift 在对象和存储节点的映射之间增加了 Partition 的概念，使得对象到存储节点的映射过程，变成了由对象到 Partition 再到存储

节点。Partition 的个数一旦确认，在整个运行过程中是不会改变的，所以对象到 Partition 的映射是不会变化的。在增加或者减少节点的情况下，通过改变 Partition 到存储节点的映射来完成数据的迁移。

而对象到 Partition 这层映射是通过哈希函数以及二进制位移操作，即这里的 Partition Shift Value 完成的，Partition 到存储节点的映射是通过设备查询表完成的。

Swift 接收到的用户请求中包含数据的路径名称，比如对于 Object 的请求可能为“GET account_name/container_name/object_name”，Swift 需要将这个路径映射到 Partition 上：

```
# swift/common/ring/ring.py

class Ring(object):
    def get_part(self, account, container=None, obj=None):
        # 获取 account/container/object 对应的 partition
        # 首先先计算哈希值
        key = hash_path(account, container, obj, raw_digest=True)
        if time() > self._mtime:
            self._reload()
        # 然后位移得到 partition
        part = struct.unpack_from('>I', key)[0] >> self._part_shift
        return part
```

3. 构建 Ring

Swift 使用 swift-ring-builder 工具构建一个 Ring。所谓构建 Ring 就是构建设备查询表的过程。构建过程大致上分为 3 个步骤。

(1) 创建 Ring 文件

```
swift-ring-builder <builder_file> create <part_power> <replicas>
<min_part_hours>
```

其中， $2^{\text{part_power}}$ 是 Partition 的个数；replicas 是副本的个数；builder_file 是 Ring 的名称；min_part_hours 单位为小时，一般设置为 24 小时，表示某个 Partition 在移动后必须等待指定的时间后才能再次移动。

(2) 添加设备到 Ring 中

```
swift-ring-builder <builder_file> add \
[r<region>] z<zone>--ip:<ip>:<port>/<device_name> <meta> <weight>
```

其中，region 和 zone 表示 Region 和 Zone 的编号；ip:port 表示该设备所在节点的 IP 地址以及提供服务的端口号；device_name 是该设备在该节点的名称（比如 sdb1）；meta 是该设备的元数据，其结构是字符串；weight 是该设备的权重。

这个步骤里，并没有任何 Partition 被实际分配到新设备上，直到执行下面的“rebalance”操作。这样做的目的是能够一次增加多个设备，然后批量地将 Partition 重新分配。

(3) 分配 Partition

```
swift-ring-builder <builder_file> rebalance
```

rebalance 操作会根据 builder file 的定义将 Partition 分配到不同的设备。在 rebalance 之后，需要把生成的 Ring 文件复制到所有运行相应服务 (Account、Container 或 Object) 的节点上，然后用该 Ring 文件作为参数启动相应的服务。

接下来我们通过源码来了解 Ring 的构建过程。通过 setup.cfg 文件可以知道 swift-ring-builder 工具的源码入口位于 bin/swift-ring-builder 脚本。这个脚本仅仅是对 swift.cli.ringbuilder 模块的封装，直接调用了 swift.cli.ringbuilder 中的 main 函数。

```
def main(arguments=None):

    if len(argv) == 2:
        command = "default"
    else:
        command = argv[2]
    if argv[0].endswith('-safe'):
        try:
            with lock_parent_directory(abspath(argv[1]), 15):
                getattr(Commands, command, Commands.unknown)()
        except exceptions.LockTimeout:
            print "Ring/builder dir currently locked."
            exit(2)
    else:
        # 调用 Commands 类的名为 "command" 的函数，如果该函数不存在则调用
        # Commands 类的 unknown() 函数，对于 Ring 的创建，"command" 为 create
        getattr(Commands, command, Commands.unknown)()
```

完成一定的参数解析等工作后，最终使用 swift.cli.ringbuilder.Commands 类的 create() 函数去完成 Ring 的创建。

```
def create():
    if len(argv) < 6:
        print Commands.create.__doc__.strip()
        exit(EXIT_ERROR)
    # 创建 RingBuilder 对象的实例
    builder = RingBuilder(int(argv[3]), float(argv[4]), int(argv[5]))
    # 为 builder 文件创建一个备份目录，该目录下会备份 builder 文件
    backup_dir = pathjoin(dirname(builder_file), 'backups')
    try:
        mkdir(backup_dir)
    except OSError as err:
        if err.errno != EEXIST:
            raise
```

```

# 将Ring的初始化信息保存到备份文件中
builder.save(pathjoin(backup_dir,
                      '%d.' % time() + basename(builder_file)))
# 将Ring的初始化信息保存到builder文件中去
builder.save(builder_file)
exit(EXIT_SUCCESS)

```

这个函数逻辑非常简单，主要就是创建一个 `swift.common.ring.builder.RingBuilder` 类的实例，然后将它的初始化信息保存到 Ring 的 builder 文件和备份文件里。

```

# swift/common/ring/builder.py

class RingBuilder(object):
    def __init__(self, part_power, replicas, min_part_hours):
        self.part_power = part_power
        self.replicas = replicas
        self.min_part_hours = min_part_hours
        self.parts = 2 ** self.part_power
        self.devs = []
        self.devs_changed = False
        self.version = 0
        self.overload = 0.0

    # _replica2part2dev 是一个二维数组，第一维从 replica 映射到 partition
    # 第二维从 partition 映射到 device。所以对一个 replica 个数为 3
    # partition 个数为 2^23 的 ring 来说，_replica2part2dev 是一个 3*2^23 数组
    # 该数组的每一个元素都是 device ID (数据类型为 unsigned short)
    self._replica2part2dev = None

    # _last_part_moves 是一个长度为 2^23 的数组，数组的每个元素为 unsigned byte
    # 这个数组的每个元素表示该元素所对应的 partition 据上次移动所过去的时间
    # (以小时为单位)。这个数组存在的目的是为了保证同一个 partition 在一定的
    # 时间内 (一般是 24 小时) 不会被移动两次。不过删除一个设备或者把一个设备
    # 的 weight 设为 0 不受这个时间的限制。这是因为删除设备或者把 weight
    # 设为 0 是因为该设备被已经出现故障。
    # _last_part_moves_epoch 表示 _last_part_moves 的基准时间
    self._last_part_moves = None
    self._last_part_moves_epoch = 0

    self._last_part_gather_start = 0
    self._dispersion_graph = {}
    self.dispersion = 0.0
    self._remove_devs = []
    self._ring = None

```

RingBuilder 类实例初始化时，在保存了传递进来的 part_power、replicas 以及 min_part_hours 等参数之后，初始化了一个重要的二维数组 _replica2part2dev。

_replica2part2dev 数组即为我们前面提到的设备查询表，它的第一维以 replica 为索引，也就是说如果设定 replicas 等于 3，那么该数组第一维就有 3 个成员，每一个成员都是一个数组（第二维数组）。第二维数组负责 partition 到 device 的映射，长度为 partition 的个数。

_replica2part2dev 之外一个重要的数组即为 devs[] 数组，该数组即为我们前面所提到的设备列表。根据从 _replica2part2dev 中检索到的设备号，到该表中查找设备的具体位置信息。目前设备表的内容为空，因为此时还不知道设备的情况。

至此，构建 Ring 的第一个步骤创建 Ring 文件已经完成，我们需要执行 swift-ring-builder 的 add 命令添加设备到 Ring 中，与 create 命令类似，add 命令由 swift.cli.ringbuilder.Commands 类的 add() 函数完成。

```
# swift/cli/ringbuilder.py

def add():
    # _parse_add_values() 解析参数，并返回一个 device 的列表，然后检查新添加的
    # device 是否已经在这个列表中。如果没有，则通过 RingBuilder 类的 add_dev()
    # 函数将新的 device 添加到 Ring 中
    for new_dev in _parse_add_values(argv[3:]):
        for dev in builder.devs:
            if dev is None:
                continue
            if dev['ip'] == new_dev['ip'] and \
                dev['port'] == new_dev['port'] and \
                dev['device'] == new_dev['device']:
                print('Device %d already uses %s:%d/%s.' %
                      (dev['id'], dev['ip'],
                       dev['port'], dev['device']))
                print("The on-disk ring builder is unchanged.\n")
                exit(EXIT_ERROR)
            dev_id = builder.add_dev(new_dev)
            print('Device %s with %s weight got id %s' %
                  (format_device(new_dev), new_dev['weight'], dev_id))

    builder.save(argv[1])
    exit(EXIT_SUCCESS)
```

最终使用第一步中创建的 swift.common.ring.builder.RingBuilder 类实例的 add_dev() 函数完成设备的添加。

```
# swift/common/ring/builder.py

def add_dev(self, dev):
```



```

"""
将一个设备添加到 ring 里面去。这个设备的 dict 数据至少需要包含以下键值 (key):
====
id      设备的唯一编号 (类型为整数)。如果 "id" key 在 dict 中没有被指定则默认
        该设备的 id 为系统中下一个可用的 id 号
weight  这个设备相对于其他设备的权重 (类型为浮点数)。这个权重用来暗示
        会有多少个 partition 分配到到这个设备上来
region  设备所在的 region 号 (类型为整数)
zone    设备所在的 zone 号 (类型为整数)。一个 partition 会被尽可能地发配到
        分布在不同的 (region, zone) 的设备上
ip      设备的 ip 地址
port    该设备的 tcp 端口
device  该设备的名字 (譬如, sdb1)
meta    元数据, 用于存储用户自定义数据, 譬如设备上线时间、硬件描述等
====

```

注意: 添加一个设备不会立即导致 rebalance, 因为用户可能想在添加多个设备之后统一做 rebalance

```

"""
if 'id' not in dev:
    dev['id'] = 0
    if self.devs:
        try:
            dev['id'] = self.devs.index(None)
        except ValueError:
            dev['id'] = len(self.devs)
    if dev['id'] < len(self.devs) and self.devs[dev['id']] is not None:
        raise exceptions.DuplicateDeviceError(
            'Duplicate device id: %d' % dev['id'])
    while dev['id'] >= len(self.devs):
        self.devs.append(None)
    dev['weight'] = float(dev['weight'])
    dev['parts'] = 0
    self.devs[dev['id']] = dev
    self.devs_changed = True
    self.version += 1
    return dev['id']

```

这个函数先计算所要新添加 device 的 ID, ID 值可以不是连续的, 设备表中间允许空洞的存在。然后将该设备加入到 Ring 的设备表中, 最后设置相关 flag, `devs_changed` 表示设备表有变化, 需要 rebalance。

这个函数返回后, `swift.cli.ringbuilder.Commands` 类的 `add()` 函数会再次调用 `RingBuilder` 类的 `save()` 函数将更新过的 Ring 信息写到 `builder` 文件中。

第二个步骤完成后，然后是执行 swift-ring-builder 的 rebalance 命令。

```
# swift/cli/ringbuilder.py

def rebalance():
    devs_changed = builder.devs_changed
    try:
        last_balance = builder.get_balance()
        # 调用 RingBuilder 类的 rebalance() 函数
        parts, balance = builder.rebalance(seed=get_seed(3))
    except exceptions.RingBuilderError as e:
        print('-' * 79)
        print("An error has occurred during ring validation. Common\n"
              "causes of failure are rings that are empty or do not\n"
              "have enough devices to accommodate the replica count.\n"
              "Original exception message:\n %s" %
              (e,))
        print('-' * 79)
        exit(EXIT_ERROR)
    if not (parts or options.force or removed_devs):
        # 没有 partition 需要移动的情况有两种，一种是经过 rebalance 的计算
        # 的确是没有任何 partition 需要移动。另外一种是由于 min_part_hours 参数
        # 的限制，在 min_part_hours 时间内只允许移动一个 partition
        print('No partitions could be reassigned.')
        print('There is no need to do so at this time')
        exit(EXIT_WARNING)
    if not options.force and \
        not devs_changed and abs(last_balance - balance) < 1 and \
        not (last_balance == MAX_BALANCE and balance == MAX_BALANCE):
        print('Cowardly refusing to save rebalance as it did not change

              'at least 1%.')
        exit(EXIT_WARNING)
    try:
        # 验证生成的 Ring 的一致性
        builder.validate()
    except exceptions.RingValidationError as e:
        print('-' * 79)
        print("An error has occurred during ring validation. Common\n"
              "causes of failure are rings that are empty or do not\n"
              "have enough devices to accommodate the replica count.\n"
              "Original exception message:\n %s" %
              (e,))
        print('-' * 79)
        exit(EXIT_ERROR)
```



```

print('Reassigned %d (%.02f%%) partitions. '
      'Balance is now %.02f. '
      'Dispersion is now %.02f' % (
          parts, 100.0 * parts / builder.parts,
          balance,
          builder.dispersion))
status = EXIT_SUCCESS

```

这个函数首先调用 `swift.common.ring.builder.RingBuilder` 类的 `get_balance()` 函数来获取当前 Ring 的 balance 值，这个值标识了一个 Ring 的平衡程度，也就是健康状况，这个值越高表示这个 Ring 越需要 rebalance。一个健康的 Ring 的 balance 的值应该是 0。

Ring 的 balance 值取决于所有 device 的 balance 值，一个 device 的 balance 值指的是超过这个 device 所希望接纳的 partition 个数的 partition 的数量，除以该 device 所希望接纳的 partition 的个数，然后乘以 100。比如一个 device 所希望接纳的 partition 的个数是 123 个，结果现在它接纳了 124 个 partition，那么这个 device 的 balance 的值就是 $(124 - 123) / 123 \times 100 = 0.83$ 。在一个 Ring 中，取所有 device 的 balance 的值得最大值作为该 Ring 的 balance 值。

如果 Ring 没有 device 的变化（添加或者删除），并且 rebalance 之前和之后的 balance 的值相差小于 1，则认为该 Ring 不需要 rebalance，不会生成新的 ring 文件。

同前两个步骤一样，rebalance 命令的实际工作仍是由 `swift.common.ring.builder.RingBuilder` 类的 `rebalance()` 来完成的。

```

# swift/common/ring/builder.py

def rebalance(self, seed=None):
    """
    这是 Ringbuilder 的主要功能函数。它会根据设备权重、zone 的信息（尽可能地将 partition 的副本分配到不在一个 zone 的设备上去），以及近期的分配情况等信息，重新对 partition 进行分配

    这个函数并不是 partition 分配的最佳方法（最佳方法会进行更多的分析从而占用更多的时间）。因此，此函数会一直做 rebalance 操作直到这个 Ring 的 balance 值小于 1%，或者 balance 的值的变化小于 1%
    """

```

6.1.3 Swift API

Swift 以 RESTful API 的形式提供自己的 API，如图 6-4 所示。Proxy Server 承担了类似 nova-api 服务的角色，负责接收并转发用户的 HTTP 请求。

Swift API 主要提供了如下功能：

- 存储对象，对象的个数并没有限制。单个对象的大小默认的最大值是 5GB，这个最大值是用户可以自行配置。

- 对于超过最大值的对象，可以通过大对象（Large Object）中间件进行上传和存储。
- 压缩对象。
- 删除对象，可以批量删除。

Swift 的对象逻辑上分为 Account、Container 和 Object 3 个层次，Swift API 也可以被分为针对 Account 的操作、针对 Container 的操作以及针对 Object 的操作，比如针对 Account 可以列出其中的所有 Container。

如果从 swiftclient 开始算起，Swift API 的执行过程主要包括几个阶段：swiftclient 将用户命令转换为标准 HTTP 请求；Paste Deploy 将请求路由到 proxy-server WSGI Application；根据请求内容调用对应 Controller（AccountController、ContainerController 或 ObjectController）处理请求，该 Controller 会将请求转发给特定存储节点上的 WSGI Server（Account Server、Container Server 或 Object Server）；Account Server、Container Server 或 Object Server 接收到 Proxy Server 转发的 HTTP 请求并处理。

第一个阶段的过程与 Nova API 相同，因此我们从第二个阶段开始介绍 Swift API 的执行过程。

1. HTTP 请求到 WSGI Application

Proxy Server 的入口是 bin/swift-proxy-server 文件：

```
from swift.common.wsgi import run_wsgi

if __name__ == '__main__':
    conf_file, options = parse_options()
    sys.exit(run_wsgi(conf_file, 'proxy-server', **options))
```

run_wsgi() 函数会启动 Proxy Server 去监听用户的 HTTP 请求，Paste Deploy 会在这个 WSGI Server 创建时参与进来，基于 Paste 配置文件/etc/swift/proxy-server.conf 去加载 WSGI Application。

在随后 swift-proxy-server 的运行过程中，Paste Deploy 会将监听到的 HTTP 请求根据 Paste 配置文件准确路由到特定的 WSGI Application。

```
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging cache
container_sync bulk tempurl ratelimit tempauth container-quotas account-quotas
slo dlo proxy-logging proxy-serve

[app:proxy-server]
account_autocreate = true
conn_timeout = 20
node_timeout = 120
use = egg:swift#proxy

[filter:slo]
```

```
use = egg:swift#slo
```

在 pipeline 里除了最后一个 proxy-server, 其他 app 都是作为 filter 的角色。“egg:swift#proxy” 表示使用 swift 包中的 proxy 模块, slo (Static Large Object) filter 定义中的 “egg:swift#slo” 表示使用 swift 包中的 slo 模块。这些模块都在 setup.cfg 文件的 “entry_points” 中进行了配置, 使用 setuptools 加载。

```
[entry_points]
paste.app_factory =
    proxy = swift.proxy.server:app_factory
    object = swift.obj.server:app_factory
    mem_object = swift.obj.mem_server:app_factory
    container = swift.container.server:app_factory
    account = swift.account.server:app_factory

paste.filter_factory =
    dlo = swift.common.middleware.dlo:filter_factory
    slo = swift.common.middleware.slo:filter_factory
.....
```

根据 setup.cfg 的配置, Paste Deploy 最终将使用 swift.proxy.server 模块的 app_factory() 函数构建 proxy-server 这个 WSGI app。

2. WSGI Application 到对应的 Controller

proxy-server 将根据请求中的信息调用相应 Controller 中的函数进行处理。与对象的 3 个层次相对应, 有 3 种 Controller: AccountController、ContainerController 和 ObjectController。这 3 种 Controller 的实现都位于 swift/proxy/controllers 目录。下面以 AccountController 为例:

```
class AccountController(Controller):
    @public
    def PUT(self, req):
        # account_ring 即为 Proxy Server 在初始化时为 Account 创建的 Ring
        # get_nodes() 返回包含该 Account 内容的 Partition
        account_partition, accounts = \
            self.app.account_ring.get_nodes(self.account_name)
        headers = self.generate_request_headers(req, transfer=True)
        clear_info_cache(self.app, req.environ, self.account_name)
        # make_requests() 会首先获得包含该 Partition 及其副本的所有节点, 然后依次
        # 将请求发送到每个节点, 直到其中一个节点返回正确的结果为止
        resp = self.make_requests(
            req, self.app.account_ring, account_partition, 'PUT',
            req.swift_entity_path, [headers] * len(accounts))
        self.add_acls_from_sys_metadata(resp)
        return resp
```

3. 存储节点上的 Account Server、Container Server 或 Object Server

用户的 HTTP 请求被 AccountController、ContainerController 与 ObjectController 分别转发给存储节点上的 Account Server、Container Server 和 Object Server。这 3 个服务与 Proxy Server 一样也是 WSGI Server，通过 `run_wsgi()` 函数启动，通过 Paste Deploy 加载对应的 WSGI Application。

(1) Account Server

Account Server 的 Paste 配置文件位于 `/etc/swift/account-server/` 目录下：

```
[pipeline:main]
pipeline = healthcheck recon account-server

[app:account-server]
use = egg:swift#account
```

类似于前面对 Proxy Server 的分析，结合 Paste 配置文件与 `setup.cfg` 文件中的设置，Paste Deploy 最终将使用 `swift.account.server` 模块的 `app_factory()` 函数加载 Account Server 的 WSGI Application，即 `swift.account.server.AccountController`。

这里的 Controller 与上述 `swift/proxy/controllers` 目录下的 Controller 不同，后者的作用是将用户的 HTTP 请求转发给 Account Server，而前者则是对该请求的最终处理。

下面以 PUT 操作为例，针对 Account 的 PUT 操作有两种语义：创建一个 Account；创建 Account 中的 Container。它们的区别在于路径参数是否包含 Container 的信息。

```
class AccountController(object):
    def PUT(self, req):
        """Handle HTTP PUT request."""
        # 从请求参数 req 里面获取 drive、partition、account 以及 container 的信息
        drive, part, account, container = split_and_validate_path(req, 3, 4)
        if self.mount_check and not check_mount(self.root, drive):
            return HTTPInsufficientStorage(drive=drive, request=req)
        # 如果 container 的信息不为空，则视该请求为创建某 account 中的 container
        if container:
            if 'x-timestamp' not in req.headers:
                timestamp = Timestamp(time.time())
            else:
                timestamp = valid_timestamp(req)
            pending_timeout = None
            container_policy_index = \
                req.headers.get('X-Backend-Storage-Policy-Index', 0)
            if 'x-trans-id' in req.headers:
                pending_timeout = 3

        # 构建并且返回一个 AccountBroker 类的实例。AccountBroker 类继承于
```



```

# DatabaseBroker 类，其内部包含针对 account 数据库文件的操作函数
# 如前所述，我们可以把 partition 理解为一个目录，每一个 partition 中
# 的 account 的数据是以这个目录中的数据库文件的形式而存在的
# 该 partition 中的每一个 account 都对应着一个数据库文件
# AccountBroker 这个类将操作 account 数据库文件的函数加以封装，
# 作为其成员函数来使用
broker = self._get_account_broker(drive, part, account,
                                   pending_timeout=pending_timeout)

# 如果该 account 的数据库文件尚未存在，则调用 AccountBroker 类的
# initialize() 函数创建该数据库文件
if account.startswith(self.auto_create_account_prefix) and \
    not os.path.exists(broker.db_file):
    try:
        broker.initialize(timestamp.internal)
    except DatabaseAlreadyExists:
        pass
if req.headers.get('x-account-override-deleted', 'no').lower() != \
    'yes' and broker.is_deleted():
    return HTTPNotFound(request=req)

# 通过调用 AccountBroker 中的 put_container() 函数将 container 的信息
# 写入该 account 的数据库文件中
broker.put_container(container, req.headers['x-put-timestamp'],
                    req.headers['x-delete-timestamp'],
                    req.headers['x-object-count'],
                    req.headers['x-bytes-used'],
                    container_policy_index)
if req.headers['x-delete-timestamp'] > \
    req.headers['x-put-timestamp']:
    return HTTPNoContent(request=req)
else:
    return HTTPCreated(request=req)
# 如果 container 的信息为空，则视该请求为创建 account
else:
    timestamp = valid_timestamp(req)

# 获取一个 AccountBroker 的实例
broker = self._get_account_broker(drive, part, account)
# 如果该 account 的数据库文件未存在，则创建
if not os.path.exists(broker.db_file):
    try:
        broker.initialize(timestamp.internal)
    created = True

```

```

except DatabaseAlreadyExists:
    created = False
elif broker.is_status_deleted():
    return self._deleted_response(broker, req, HTTPForbidden,
                                   body='Recently deleted')
else:
    created = broker.is_deleted()
    broker.update_put_timestamp(timestamp.internal)
    if broker.is_deleted():
        return HTTPConflict(request=req)

metadata = {}
metadata.update((key, (value, timestamp.internal))
                 for key, value in req.headers.iteritems()
                 if is_sys_or_user_meta('account', key))
if metadata:
    # 更新元数据
    broker.update_metadata(metadata)
if created:
    return HTTPCreated(request=req)
else:
    return HTTPAccepted(request=req)

```

(2) Container Server

类似于 Account Server, Paste Deploy 最终将使用 swift.container.server 模块的 `app_factory()` 函数加载 Container Server 的 WSGI Application, 即 `swift.container.server.ContainerController`。

下面以 GET 操作为例:

```

class ContainerController(object):
    def GET(self, req):
        """Handle HTTP GET request."""
        # 从请求参数 req 中获取 drive、partition、account、container
        # 以及 object 的信息
        drive, part, account, container, obj = split_and_validate_path(
            req, 4, 5, True)
        # prefix、delimiter、marker、end_marker 可以作为查询 object 的条件
        # 比如可以利用 prefix 参数查询前缀为某个字符串的 object
        path = get_param(req, 'path')
        prefix = get_param(req, 'prefix')
        delimiter = get_param(req, 'delimiter')
        marker = get_param(req, 'marker', '')
        end_marker = get_param(req, 'end_marker')
        limit = constraints.CONTAINER_LISTING_LIMIT
        given_limit = get_param(req, 'limit')

```



```

out_content_type = get_listing_content_type(req)

# 获取一个 ControllerBroker 类的实例。与 Account Server 类似
# Container 的相关信息也是作为一个 sqlite 数据库文件存放于相应 partition 的
# 目录下的。ControllerBroker 类封装了对该数据库文件进行访问的方法
broker = self._get_container_broker(drive, part, account, container,
                                     pending_timeout=0.1,
                                     stale_reads_ok=True)

# 判断是否被删除
info, is_deleted = broker.get_info_is_deleted()
resp_headers = gen_resp_headers(info, is_deleted=is_deleted)
if is_deleted:
    return HTTPNotFound(request=req, headers=resp_headers)

# 调用 ContainerBroker 类的 list_objects_iter() 函数读取 container 数
# 据库文件, 返回 object 信息的列表
container_list = broker.list_objects_iter(
    limit, marker, end_marker, prefix, delimiter, path,
    storage_policy_index=info['storage_policy_index'])
return self.create_listing(req, out_content_type, info, resp_headers,
                           broker.metadata, container_list, container)

```

(3) Object Server

同样, Paste Deploy 最终将使用 swift.obj.server 模块的 `app_factory()` 函数加载 Object Server 的 WSGI Application, 即 `swift.obj.server.ObjectController`。

下面以 DELETE 操作为例:

```

class ObjectController(object):
    def DELETE(self, request):
        """Handle HTTP DELETE requests for the Swift Object Server."""
        # 请求参数 req 中获取 device、partition、account、container、object
        # 以及 storage policy 的相关信息
        device, partition, account, container, obj, policy = \
            get_name_and_placement(request, 5, 5, True)
        req_timestamp = valid_timestamp(request)
        try:
            # Swift 将 object 在磁盘上的二进制数据抽象为一个 DiskFile 类, 该类封装了
            # 创建、删除以及读写 object 等方法。而且可以通过使用不同的 DiskFile 类
            # 来达到实现不同 Object Server 的目的
            # 此处通过调用 get_diskfile() 方法获取一个 DiskFile 类的实例
            disk_file = self.get_diskfile(
                device, partition, account, container, obj,
                policy=policy)
        except DiskFileDeviceUnavailable:

```

```

        return HTTPInsufficientStorage(drive=device, request=request)
    try:
        # 读取元数据
        orig_metadata = disk_file.read_metadata()
    except DiskFileXattrNotSupported:
        return HTTPInsufficientStorage(drive=device, request=request)
    except DiskFileExpired as e:
        # 过期处理
        orig_timestamp = e.timestamp
        orig_metadata = e.metadata
        response_class = HTTPNotFound
    except DiskFileDeleted as e:
        # 已经被删除
        orig_timestamp = e.timestamp
        orig_metadata = {}
        response_class = HTTPNotFound
    except (DiskFileNotExist, DiskFileQuarantined):
        # 不存在或者被 auditor 隔离 (数据损坏)
        orig_timestamp = 0
        orig_metadata = {}
        response_class = HTTPNotFound
    else:
        orig_timestamp = Timestamp(orig_metadata.get('X-Timestamp', 0))
        if orig_timestamp < req_timestamp:
            response_class = HTTPNoContent
        else:
            response_class = HTTPConflict
    response_timestamp = max(orig_timestamp, req_timestamp)

    # Swift 为 object 提供了名为 X-Delete-At 的 metadata。X-Delete-At
    # 的意思是如果到了“X-Delete-At”所表示的时间戳，则将 object 删除
    # 这个功能主要是为 object-expirer 准备的。object-expirer 是一个后台程序
    # 会定期检查，并且删掉那些已过期的 object
    # “x-if-delete-at”参数的意思是如果“X-Delete-At”的值等于
    # “x-if-delete-at”则删掉该 object
    orig_delete_at = int(orig_metadata.get('X-Delete-At') or 0)
    try:
        req_if_delete_at_val = request.headers['x-if-delete-at']
        req_if_delete_at = int(req_if_delete_at_val)
    except KeyError:
        pass
    except ValueError:
        return HTTPBadRequest(
            request=request,

```

```

        body='Bad X-If-Delete-At header value')
    else:
        if not orig_timestamp:
            return HTTPNotFound()
        if orig_delete_at != req_if_delete_at:
            return HTTPPreconditionFailed(
                request=request,
                body='X-If-Delete-At and X-Delete-At do not match')
        else:
            response_class = HTTPNoContent
    if orig_delete_at:
        # 更新 container 的 delete_at 信息
        self.delete_at_update('DELETE', orig_delete_at, account,
                               container, obj, request, device,
                               policy)
    if orig_timestamp < req_timestamp:
        # 删除 object 二进制文件, 这里并没有真正地将文件删除, 而是创建一个
        # 后缀为.ts (tombstone) 的文件作为这个 object 的最新版本, 后续由
        # replicator 来做真正的删除操作。Replicator 是后台程序, 不需要占用
        # DELETE 操作的时间
        try:
            disk_file.delete(req_timestamp)
        except DiskFileNoSpace:
            return HTTPInsufficientStorage(drive=device, request=request)
        # 更新 container
        self.container_update(
            'DELETE', account, container, obj, request,
            HeaderKeyDict({'x-timestamp': req_timestamp.internal}),
            device, policy)
    return response_class(
        request=request,
        headers={'X-Backend-Timestamp': response_timestamp.internal})

```

6.1.4 认证

Swift 通过 Proxy Server 接收用户 RESTful API 请求时, 首先需要通过认证服务对用户的身份进行认证, 认证通过后, Proxy Server 才会真正地处理用户请求并响应。

Swift 支持外部和内部两种认证方式。一般来说, 外部的认证是指向 Keystone 服务去认证, 内部的认证是指通过 Swift 的 WSGI 中间件 Tempauth 来认证用户。无论通过何种方式, 用户首先需要提交自己的 credentials 给认证系统, 认证系统会返回给用户一个文本形式的 token。这个 token 有一定的时效性, token 验证的结果会被缓存, 用户可以在 token 尚未过期的时间内通过在请求中指定 token 来访问 Swift 服务。

具体使用何种认证方法在 Proxy Server 的 Paste Deploy 配置文件/etc/swift/proxy-server.conf 中进行设置。

```
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging cache
container_sync bulk tempurl ratelimit tempauth container-quotas account-quotas
slo dlo proxy-logging proxy-serve
```

Swift 默认采用 Tempauth 认证方式。因为是采用 WSGI 中间件的形式，我们可以很容易地实现一个自己的认证服务替换 Keystone 或 Tempauth。下面以 Keystone 为例，Swift 认证过程如图 6-7 所示。

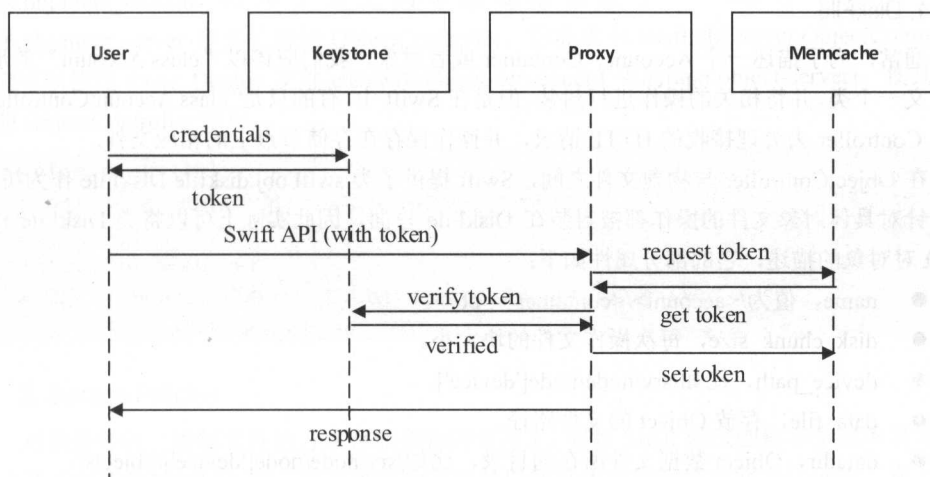


图 6-7 Swift 认证过程

如果使用 Tempauth，需要在 proxy-server.conf 的 tempauth 部分定义用户的信息，其格式如下：

```
user_<account>_<user> = <key> [group] [group] [...] [storage_url]
```

key 就是密码。group 有两种，一种是.reseller_admin，具有对任何 Account 操作的权限；另外一种是.admin，只能对所在 Account 进行操作。如果没有设置这两种 group 的任何一个，则该用户只能访问那些.admin 与.reseller_admin 所允许的 Container。最后的 storage_url 用于在认证之后向用户返回 Swift 的 URL。以下是 proxy-server.conf.sample 中提供的例子。

```
user_admin_admin = admin .admin .reseller_admin
user_test_tester = testing .admin
user_test2_tester2 = testing2 .admin
user_test_tester3 = testing3
```

如果希望使用 keystone 认证, 那么需要在 pipeline 中指定 authtoken 以及 keystoneauth 中间件, 并且 authtoken 需要排在 keystoneauth 之前, 同时也需要在 proxy-server.conf 中做相应配置。

6.1.5 对象管理与操作

我们已经知道 Swift 通过 Account、Container 与 Object 3 个层次进行对象的组织与管理, 同时对象最终将以二进制文件的形式存储在物理的存储节点上, 那么这里的问题就是 Swift 如何去描述一个对象, 并将抽象的对象与实际的具体文件联系起来。

1. DiskFile

通常, 为了描述一个 Account、Container 或者对象, 我们应该以 “class Account” 的形式去定义一个类, 并将相关的操作进行封装, 但是在 Swift 中, 有的只是 “class AccountController”, 各个 Controller 去处理接收的 HTTP 请求, 并操作保存在存储节点上的相应文件。

在 ObjectController 与物理文件之间, Swift 提供了类 swift.obj.diskfile.DiskFile 作为桥梁, 所有针对具体对象文件的操作都被封装在 DiskFile 里面, 因此实际上可以将类 DiskFile 作为 Swift 对对象的描述, 它的部分属性如下:

- name, 值为 /<account>/<container>/<obj>。
- disk_chunk_size, 每次操作文件的块大小。
- device_path, 比如 /srv/node/node['device']。
- data_file, 存放 Object 的文件路径。
- datadir, Object 数据文件所在的目录, 比如 /srv/node/node['device']/objects/。
- metadata, 对象的元数据。

它所封装的文件操作都对应了针对对象的 RESTful API, 如表 6-1 所示。

表 6-1 Object RESTful API

方 法	URI	描 述
GET	/v1/{account}/{container}/{object} {?signature,expires,multipart-manifest}	下载 object 的内容并且获取该 object 的 metadata
PUT	/v1/{account}/{container}/{object} {?multipart-manifest,signature,expires}	用传递进来的数据内容以及 metadata 创建或者替换一个 object
COPY	/v1/{account}/{container}/{object}	复制一个 object
DELETE	/v1/{account}/{container}/{object} {?multipart-manifest}	永久性删除一个 object
HEAD	/v1/{account}/{container}/{object} {?signature,expires}	获取 object 的 metadata
POST	/v1/{account}/{container}/{object}	创建或者更新 object 的 metadata

但是, 不同的存储介质不同的文件系统对于文件的操作方式可能会有些差异, 我们无法用一个类 DiskFile 涵盖所有的情况, 为了支持不同的存储后端 (storage backend), Swift 引入

了 PBE (pluggable backends, 可插拔后端, 亚特兰大 Summit 上 Swift 的热门话题之一, <http://techs.enovance.com/6981/openstack-swift-atlanta-summit-retro>) 的概念。

PBE 通过实现特定的类 `DiskFile` 去支持新的存储后端, 因为 `ObjectController` 负责响应 RESTful API 并通过类 `DiskFile` 进行具体的文件操作, 所以所有的类 `DiskFile` 实现必须要满足 `ObjectController` 处理流程的需要, 官方文档中的 “Back-end API for Object Server REST APIs” (http://docs.openstack.org/developer/swift/development_ondisk_backends.html#back-end-api-for-object-server-rest-apis) 给出了需要实现的接口的详细描述, 比如所有的 `DiskFile` 必须实现对象内容和元数据的读写。

Swift 提供了一个简单的示例, 实现了一个内存文件系统的后端接口: `swift.obj.mem_diskfile` 按照上面文档中的要求实现了一个新的 `DiskFile` 类, `swift.obj.mem_server` 定义了新的 `ObjectController`, 它继承于 `swift.obj.server.ObjectController`, 我们可以修改 `Paste Deploy` 文件 `/etc/swift/object-server.conf` 中的 `[app:object-server]`, 使其使用新的 `ObjectController` 即可。

```
[pipeline:main]
pipeline = healthcheck recon object-server

[app:object-server]
# 默认为 object, 使用 swift.obj.server.ObjectController
use = egg:swift#mem_object
```

2. Storage Policies

对象最终以二进制文件的方式存储在物理节点上, 并且 Swift 通过创建多个副本等冗余技术达到了极高的数据持久性, 但是副本的采用是以牺牲更多的存储空间为代价的, 那么这里的另外一个问题就是能否通过其他的技术来减少存储空间的占用。

Swift 在 Kilo 版本中实现了 EC (Erasure Coding) 技术来减少存储空间。EC 技术将数据分块, 再对每一块加以编码, 从而减少对存储空间的需求, 并且还可以在某一块数据被损坏的情况下根据其他块的数据将其恢复。其实是通过消耗更多计算和网络带宽资源来减少对存储资源的消耗。

为了能够让 EC 和现有的基于副本的实现并存, `Storage Policies` 应运而生 (http://docs.openstack.org/developer/swift/overview_policies.html)。一个 `Storage Policy` 可以简单理解为一种存储方式或策略, 比如要求为每一个 `Partition` 创建两个副本。

通过为每一个 `Storage Policy` 配备一个 `Object Ring`, Swift 实现了对采用不同 `Storage Policy` 的 `Object` 采取不同的存储方式。

EC 的实现在 Kilo 中还只是作为 beta 版本发布, 但是可以说 EC 是 `Storage Policies` 提出的主要原因和动力, `Storage Policies` 也被设计成一种通用的实现。

举例来说, 一个 Swift 的部署可能会存在这样两个 `Storage Policy`: 一个要求每一个 `Partition`

都有 3 个副本，另外一个只要求有两个副本，后者服务级别比较低。另外还可以存在一个 Storage Policy 包含 SSD 硬件设备，从而使得应用 Storage Policy 的用户都能得到较高的存储效率。

使用 Storage Policies 的核心问题就是如何确定一个 Object 的 Storage Policy。我们知道 Swift 按照 Account、Container 和 Object 3 个层次来组织对象，一个新创建的 Object 必然包含在一个 Container 中。Swift 要求每一个 Container 都有和它相关联的一个 Storage Policy。这种关联是多对一的，也就是说，多个 Container 可以关联到同一个 Storage Policy 上。这种关联关系在 Container 创建时确立，并且不可改变。这样，在某一个 Container 里面创建的 Object 都将采用这个 Container 所关联的 Storage Policy。

我们可以通过/etc/swift.conf 文件来配置 Storage Policies：

```
# Storage Policies 指定了关于如何存储和对待 Object 的一些属性。每一个 Container
# 都与一个 Storage Policy 相关联。这种关联方式是通过为每一个 Container 都指定一个
# Storage Policy 的名字来实现的。Storage Policy 的名字区分大小写字母
# Storage Policy 的索引(index)在配置文件中每一个 Storage Policy section 的 header
# 部分指定。索引被内部代码所使用
# 索引为 0 的 Storage Policy 预留给在 Storage Policy 出现之前创建的 Container 使用
# 可以为索引为 0 的 Storage Policy 指定一个名字以便在元数据中使用
# 但是索引为 0 的 Storage Policy 的 Ring 文件的名字永远是“object.ring.gz”，这是
# 为了兼容在 Storage Policy 出现之前创建的 Container
# 如果没有指定 Storage Policy，那么一个名为“Policy-0”，索引为 0 的 Storage Policy
# 会被自动创建
# 使用“default”关键字指定默认的 Storage Policy。在创建新的 Container 时如果
# 没有指定 Storage Policy 那么就使用默认的 Storage Policy 与其关联
# 如果没有指定默认的 Storage Policy，则将索引为 0 的 Storage Policy 视为默认值
# 如果创建了多个 Storage Policy 则必须指定一个索引为 0 的 Storage Policy 以及一个
# 默认的 Storage Policy

# storage-policy:0.
[storage-policy:0]
name = Policy-0
default = yes

# 下面的 section 示范了如何创建一个名字为“silver”的 Storage Policy
# 每一个 Storage Policy 都有一个 Object Ring，创建这个 Ring 时所指定
# 的副本个数也就是这个 Storage Policy 的副本个数
# 在这个例子中，“silver”可以有一个比上述“Policy-0”高的或者低的副本数量
# 这个 Storage Policy 的 Ring 文件名字是“object-1.ring.gz”
# 如果把“silver”作为默认的 Storage Policy，那当一个 Container 被创建时
# 如果没有指定 Storage Policy，那么这个 Container 就会与“silver”相关联
# 但是如果 Swift 访问的是一个在 Storage Policy 出现之前创建的 Container，那么
# 该 Container 所关联的依然是索引号为 0 的 Storage Policy
```

```
#[storage-policy:1]
#name = silver
```

下面以 Policy-0 为例, “[storage-policy:0]” 说明 Storage Policy 的索引 (index) 是 0。Storage Policies 的内部实现是用索引而非名字来检索。

“name = Policy-0” 说明 Storage Policy 的名字叫做 “Policy-0”。

“default = yes” 说明 Storage Policy 是默认的 Storage Policy。

我们看到在定义 “Policy-0” 之后, 在注释里面又定义了一个名字为 “silver”、索引号为 1 的 Storage Policy。

Storage Policy 和 Object Ring 之间的 1:1 映射是通过索引号来建立的。索引号为 0 的 Storage Policy 所对应的 Ring 文件的名字为 object.ring.gz。索引号为 1 的 Storage Policy 所对应的 Ring 文件的名字为 object-1.ring.gz, 依此类推。

那么如何在创建一个 Container 的时候指定使用何种 Storage Policy 呢? 这通过一个特殊的 Request header——“X-Storage-Policy” 来实现。如果通过 “X-Storage-Policy” 指定了所使用的 Storage Policy 的名字, 那么 Container 就和该 Storage Policy 相关联, 否则就关联到默认的 Storage Policy 上。

Storage Policy 的数据结构为类 `swift.common.storage_policy.StoragePolicy`, 我们并不需要通过实例化类 `StoragePolicy` 来创建 Storage Policy, 而是推荐使用 `swift.common.storage_policy.reload_storage_policies()` 函数从 Swift 配置文件 `etc/swift/swift.conf` 中加载 Storage Policy。

`StoragePolicy` 类最重要的成员就是 `object_ring`, 也就是 Storage Policy 所对应的 Object Ring。`object_ring` 既可以在初始化的时候作为参数传进来, 也可以通过调用 `load_ring()` 函数从一个 Ring 文件中读取出来。

那么写在 Swift 配置文件中的 Storage Policy 又是什么时候被加载到 Swift 的运行系统去的呢? 这个操作在 Proxy Server 中完成。`swift.proxy.server` 从 `swift.common.storage_policy` 中 import 了全局变量 `POLICIES`, 并且在 `swift.proxy.server.Application` 类的 `__init__()` 函数中进行加载:

```
# ensure rings are loaded for all configured storage policies
for policy in POLICIES:
    policy.load_ring(swift_dir)
```

其实质就是为每一个 Storage Policy 加载相应的 Ring。

6.1.6 数据一致性

目前为止, 我们所主要介绍的都只是如何在磁盘上存储数据并向用户提供 RESTful API, 看起来这并不是难以解决的问题, 但是为了能够应用于实际的云环境, Swift 必须得考虑应该如何面对数据的损坏或硬件的故障等问题。

Swift 通过为对象引入多个副本来保障一个数据的损坏或部分硬件的故障不会引起该数

据的丢失，并通过 Sotrage Policies 来减轻多个副本所带来的存储资源消耗，但是由此却引入了另外一个问题，同一对象多个副本之间的一致性如何得以保证。

1. NWR 策略

Swift 保证数据一致性的理论依据是 NWR 策略（又称为 Quorum 仲裁协议），其中， N 为数据的副本总数， W 为更新一个数据对象时需要确保成功更新的份数， R 为读取一个数据时需要读取的副本个数。

如果 $W + R > N$ ，那么就可以保证某个数据不能被两个不同的事务同时读写。否则，如果有两个事务同时对同一数据进行读写，那么在 $W + R > N$ 的情况下，必然会有至少一个副本发生读写冲突。

如果 $W > N/2$ ，那么可以保证两个事务不能并发写同一个数据，否则，必然会有至少一个副本发生写冲突。

既然 Swift 使用了多个副本来保证数据的高持久性，那么 N 必须大于 1，如果 N 为 2，那么只要有一个数据损坏或存储节点故障，就会有数据单点的存在。一旦这个数据再次出错，就可能永久地丢失，所以 N 应该大于 2。但是 N 越高，系统的整体成本也就越高，所以 Swift 默认采用了 $N=3$ ， $W=2$ ， $R=2$ 的设置，表示一个对象默认有 3 个副本，至少需要更新两个副本才算写成功，至少读两个副本才算读成功。如果 $R=1$ ，则可能会读取到旧版本的数据。

2. Auditor、Updater 与 Replicator

有时同一数据的各个副本之间会出现不一致的情况，比如更新一个 Object 时，依照 NWR 策略，只要有两个副本更新成功，这个更新操作就被认为是成功的，剩下的那个没有更新成功的副本就会与其他两个副本不一致。那么就需要有一种机制来保证各个副本之间的一致性。

Swift 中引入了 3 种后台进程来解决数据的一致性问题：Auditor、Updater 和 Replicator。Auditor 负责数据的审计，通过持续的扫描磁盘来检查 Account、Container 和 Object 的完整性，如果发现数据有所损坏，Auditor 就会对文件进行隔离，然后从其他节点上获取一份完好的副本来取代，而这个副本的任务则由 Replicator 来完成。此外，前面已经提及，在 Ring 的 rebalance 操作中，需要 Replicator 来完成实际的数据迁移工作，在 Object 删除的时候，也是由 Replicator 来完成实际的删除操作。

Updater 负责处理那些因为负荷不足等原因而失败的 Account 或 Container 更新操作。Updater 会扫描本地节点上的 Container 或 Object 数据，然后检查相应的 Account 或 Container 节点上是否存在这些数据的记录。如果没有的话，则将这些数据的记录推送到该 Account 或 Container 节点上。只有 Container 和 Object 有对应的 Updater 进程，并不存在 Account 的 Updater 进程。

这 3 种进程的实现过程类似，这里只是以 Account 的 Replicator 进程为例。Swift 中存在着两种 Replicator：一种是 Database Replicator，针对的是 Account 和 Container 这两种以数据

库形式存在的数据；另外一种 Object Replicator，服务于 Object 数据。

Account 的 Replicator 进程起点为 bin/swift-account-replicator，工作流程中的关键部分如下所述。

- 使用 `swift.common.daemon.run_daemon()` 创建后台进程。

与 Account Server、Container Server、Object Server 与 Proxy Server 等通过 `run_wsgi()` 函数来启动 WSIG Server 不同，Swift 的其他后台进程都是使用 `run_daemon()` 函数来创建的。

对 Account 的 Replicator 进程来说，它对应的实现类 `swift.account.replicator.AccountReplicator` 继承自类 `swift.common.db_replicator.Replicator` (Database Replicator 的基类，ContainerReplicator 也继承自这个类)，而这个类又是 `swift.common.daemon.Daemon` 的子类，并实现了 `run_once()` 以及 `run_forever()` 等方法来完成数据库文件的复制。

- Replicator 进程的主要工作由 `run_once()` 完成。

```
def run_once(self, *args, **kwargs):
    # 从 Ring 上获取所有设备，遍历并判断是否为本地设备，如果是则将该设备
    # 对应的 datadir = /srv/node/node['device']/accounts 和 node['id']
    # 作为元素储存在字典 dirs 中
    for node in self.ring.devs:
        if node and is_local_device(ips, self.port,
                                    node['replication_ip'],
                                    node['replication_port']):
            found_local = True
            if self.mount_check and not ismount(
                os.path.join(self.root, node['device'])):
                self._add_failure_stats(
                    [(failure_dev['replication_ip'],
                        failure_dev['device'])
                     for failure_dev in self.ring.devs if failure_dev])
                self.logger.warning(
                    _('Skipping %(device)s as it is not mounted') % node)
                continue
            unlink_older_than(
                os.path.join(self.root, node['device'], 'tmp'),
                time.time() - self.reclaim_age)
            datadir = os.path.join(self.root, node['device'], self.datadir)
            if os.path.isdir(datadir):
                self._local_device_ids.add(node['id'])
                dirs.append((datadir, node['id']))
    if not found_local:
        self.logger.error("Can't find itself %s with port %s in ring "
                          "file, not replicating",
                          ", ".join(ips), self.port)
    self.logger.info(_('Beginning replication run'))
```



```

# 遍历 node['device']/accounts 下的每一个文件 object_file(这个目录中具体
# partation 下以.db 为后缀的文件, 比如 partition/suffix/*.db), 并调用
# _replicate_object() 复制本地指定 partation 中的数据到指定节点, 从而
# 实现各个副本之间的同步
for part, object_file, node_id in roundrobin_datadirs(dirs):
    self.cpool.spawn_n(
        self._replicate_object, part, object_file, node_id)
self.cpool.waitall()
self.logger.info(_('Replication run OVER'))
self._report_stats()

```

- 具体的复制逻辑由 `_replicate_object()` 完成。

`_replicate_object()` 首先获取该 Partation 所在的所有存储节点, 并依次向这些目标节点发送 HTTP REPLICATE 复制请求, 实现本地文件到远程指定节点的同步操作 (采用 push 模式, 而不是 pull 模式)。

收到响应后, 通过比较 Hash 值和同步点来判断复制后的两个副本是否一致, 即复制操作是否成功。如果判断不成功, 则需要比较两个副本的差异程度。如果超过 50%, 即意味着差异比较大, 则通过命令 `rsync` 实现全部数据的同步。否则, 只是发送自上一次同步以来的所有数据变化来实现两个副本的一致。

6.2 Cinder

Cinder 前身是 Nova 中的 nova-volume 服务, 在 Folsom 版本发布时, 从 Nova 中剥离作为一个独立的 OpenStack 项目存在。

6.2.1 Cinder 体系结构

与 Nova 利用主机本地存储为虚拟机提供的临时存储不同, Cinder 则类似于 Amazon 的 EBS(Elastic Block Storage), 为虚拟机提供持久化的块存储能力, 实现虚拟机存储卷(Volume) 的创建、挂载卸载、快照(Snapshot) 等生命周期管理。

不同于 Swift 在存储数据与具体存储设备和文件系统之间引入了“对象”的概念作为一层抽象, Cinder 则是在虚拟机与具体存储设备之间引入了一层“逻辑存储卷”的抽象, 因此, Swift 提供的 RESTful API 主要是用于对象的访问, Cinder 提供的 RESTful API 则主要是针对逻辑存储卷的管理。图 6-8 所示为 Cinder 的架构。

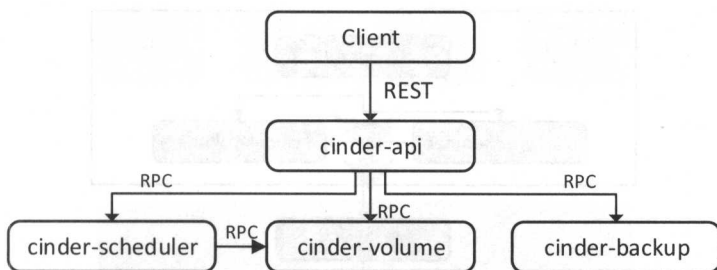


图 6-8 Cinder 架构

由图 6-8 可以看出，目前的 Cinder 主要由 cinder-api、cinder-scheduler、cinder-volume 以及 cinder-backup 几个服务所组成，它们之间通过 AMQP 消息队列进行通信。

- cinder-api 是进入 Cinder 的 HTTP 接口。
- cinder-volume 运行在存储节点上管理具体存储设备的存储空间，每个存储节点上都会运行一个 cinder-volume 服务，多个这样的节点一起便构成了一个存储资源池。
- cinder-scheduler 会根据预定的策略（比如不同的调度算法）选择合适的 cinder-volume 节点来处理用户的请求。在用户的请求没有指定具体的存储节点时，会使用 cinder-scheduler 选择一个合适的节点。如果用户请求已经指定了具体的存储节点，则该节点上的 cinder-volume 会进行处理，并不需要 cinder-scheduler 的参与。
- cinder-backup 用于提供存储卷的备份功能，支持将块存储卷备份到 OpenStack 备份存储后端，比如 Swift、Ceph、NFS 等。

如前所述，Cinder 在虚拟机与具体存储设备之间引入了“逻辑存储卷”的抽象，但 Cinder 本身并不是一种存储技术，并没有实现对块设备的实际管理和服务。它只是提供了一个中间的抽象层，为后端不同的存储技术，比如 DAS、NAS、SAN、对象存储以及分布式文件系统等，提供了统一的接口，不同的块设备服务厂商在 Cinder 中以驱动的形式实现这些接口来与 OpenStack 进行整合。Wiki 页面 <https://wiki.openstack.org/wiki/CinderSupportMatrix> 上列举了包括 NetAPP、IBM、华为、EMC 在内的众多存储厂商以及很多开源块存储系统对 Cinder 的支持细节。更为细化的 Cinder 架构如图 6-9 所示。

Cinder 默认使用 LVM（Logical Volume Manager）作为后端存储（Backend Storage），它由 Heinz Mauelshagen 于 Linux 2.4 内核实现。

通常我们在 Linux 系统里使用 fdisk 工具来分割并管理磁盘的分区，比如将磁盘/dev/sda 分割为/dev/sda1 与/dev/sda2 两个分区分别满足不同的需要，但是这种手段非常生硬，比如需要重新引导系统来使分区生效。

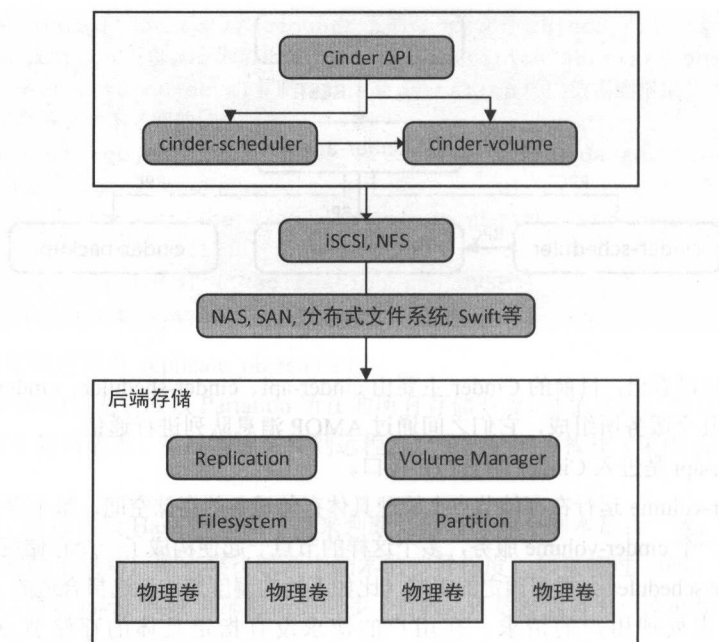


图 6-9 更为细化的 Cinder 架构

而 LVM 通过在操作系统与物理存储资源之间引入逻辑卷（Logical Volume）的抽象来解决传统磁盘分区管理工具的问题。LVM 将众多不同的物理存储器资源（物理卷，Physical Volume，比如磁盘分区）组成卷组。卷组可以理解成普通系统中的物理磁盘，但是卷组上并不能创建或安装文件系统，而是需要 LVM 从卷组中创建一个逻辑卷，然后将 ext3、ReiserFS 等文件系统安装在这个逻辑卷上，我们可以在不重新引导系统的前提下，通过在卷组里划分额外的空间为这个逻辑卷动态扩容。

比如，如图 6-10 所示由 4 个磁盘分区组成的 LVM 系统，LVM 在由这 4 个磁盘分区组成的卷组上创建了多个逻辑卷作为逻辑分区。如果需要为一个逻辑分区扩充存储空间，只需从剩余空间上分配一些给该逻辑分区使用。

除了 LVM，目前 Cinder 以驱动的形式已经支持众多存储技术或存储厂商的设备作为后端存储，比如 SAN（Storage Area Network）、Sheepdog，以及 EMC、华为等厂商的设备。

SAN 采用 FC（Fibre Channel，光线通道）技术，通过 FC 交换机连接存储阵列和服务器主机，建立专用于数据存储的区域网络。但是 FC 设备价格比较昂贵，为了降低成本，SAN 可以使用基于 IP 协议的 iSCSI 协议建立，并不受 SCSI 的布局限制。SCSI 通常要求设备互靠近并且使用 SCSI 总线连接，而 iSCSI 可适用于服务器主机和存储设备在 TCP/IP 网络上进行大量数据的可靠传输。

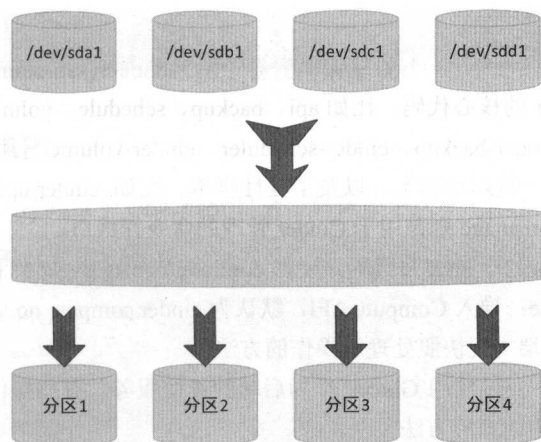


图 6-10 LVM 示例

Sheepdog 是一个类似于 Ceph 的分布式存储系统开源实现，由 NTT 的 3 名日本研究员开发，淘宝也是 Sheepdog 社区的主要贡献者。

1. Cinder 源码目录结构

```

├── api-ref
├── cinder
│   ├── api
│   ├── backup
│   ├── brick
│   ├── cmd
│   ├── common
│   ├── compute
│   ├── consistencygroup
│   ├── db
│   ├── hacking
│   ├── image
│   ├── keymgr
│   ├── locale
│   ├── objects
│   ├── replication
│   ├── scheduler
│   ├── tests
│   ├── transfer
│   ├── volume
│   └── zonemanager
└── etc

```

```
├── setup.cfg
└── setup.py
```

- **cinder**: cinder 的核心代码。比如 **api**、**backup**、**schedule**、**volume** 几个子目录分别是 **cinder-api**、**cinder-backup**、**cinder-scheduler**、**cinder-volume** 等服务的具体实现。
- **cinder/cmd**: 一些启动脚本，以及工具性脚本。比如 **cinder-api** 负责启动 **cinder-api** 服务，**cinder-manage** 则是用于 Cinder 管理的命令行接口。
- **cinder/common**: 一些公共代码，比如 **common/config.py** 定义了一些配置参数信息。
- **cinder/compute**: 导入 Compute API，默认为 **cinder.compute.nova.API**，定义了一些通过 Nova 客户端实现快照处理等操作的方法。
- **cinder/image**: 实现使用 Glance 作为后端的镜像服务，有些操作通过 Glance 客户端调用 Glance 中的相应方法实现。
- **cinder/keymgr**: 用于密钥管理。
- **cinder/replication**: 管理卷的副本。卷的副本是一个对 HA (High Availability) 和容灾恢复 (Disaster Recovery) 相当关键的存储功能。详见相关的 Cinder spec (<https://github.com/openstack/cinder-specs/blob/master/specs/juno/volume-replication.rst>)。
- **cinder/transfer**: 处理卷所有权转换相关的请求，比如卷从一个租户转换到另一个租户。
- **cinder/zonemanager**: 扩展 Cinder 对 FC 的支持。
- **etc**: 配置文件模板，包括 Paste 配置文件等。

2. setup.cfg

依照惯例，理解具体的实现之前，我们需要仔细浏览 **setup.cfg** 文件。

```
console_scripts =
    cinder-all = cinder.cmd.all:main
    cinder-api = cinder.cmd.api:main
    cinder-backup = cinder.cmd.backup:main
    cinder-manage = cinder.cmd.manage:main
    cinder-rootwrap = oslo_rootwrap.cmd:main
    cinder-rtstool = cinder.cmd.rtstool:main
    cinder-scheduler = cinder.cmd.scheduler:main
    cinder-volume = cinder.cmd.volume:main
    cinder-volume-usage-audit = cinder.cmd.volume_usage_audit:main
```

类似于 Swift，对于 Cinder 来说，**setup.cfg** 文件中值得关注的是 **console_scripts** 关键字所对应的内容，其中的每一项都代表了一个被安装在系统里的可执行脚本，它们同时也是 Cinder 各项工作的入口，完全可以作为我们理解 Cinder 具体实现的起点。

- **cinder-all**: 在一个进程里启动所有的 Cinder 服务。
- **cinder-rtstool**: 伴随 LIO (Linux-IO Target, SCSI Target 的开源实现) 支持而增加的

工具。

- `cinder-volume-usage-audit`: 用于卷使用情况统计。

6.2.2 Cinder API

Cinder API 相关源码位于 `cinder/api` 目录，具体如下：

```
.
├── contrib
├── middleware
├── openstack
├── v1
├── v2
├── v3
└── views
```

在第 5 章介绍 Nova API 时，我们提到 Nova 中每一个 API 都对应了一种资源。Nova 资源被划分为核心资源与扩展资源，扩展资源根据具体实现的不同又包含其他资源的扩展，或者自己本身就是一种新的资源两种情况。

对于 Cinder，我们同样可以如此理解，`contrib` 目录下存放的即是所有的扩展资源，而核心资源的实现则又有 `v1`、`v2` 与 `v3` 这 3 个版本，分别位于 `v1`、`v2` 与 `v3` 目录。这些 API 的实现主要涵盖了对 Volume、Volume 类型（Volume Type）以及 Snapshot 的管理操作。

Volume 类型是用户自定义的卷的一种标识。Cinder 提供了相关的 API 可以自由地创建删除 Volume 类型。

Snapshot 是一个 Volume 在某一个特定时间点的一个快照，因此，Snapshot 是只读的，不可以被改变的。Snapshot 可以被用来创建一个新的 Volume。

与 Nova 不同的是，Nova 的 `v3` 和 `v2.1` API 对应的所有资源作为插件在 `setup.cfg` 的 `entry_points` 中进行了配置，使用 `stevedore` 进行加载，而 Cinder API 无论是 `v1`、`v2` 还是 `v3` 都与 Nova API 的 `v1` 版本比较类似，扫描 `contrib` 目录发现所有的资源进行依次加载。

但是在加载所有的扩展资源时，Cinder 根据配置文件 `/etc/cinder/cinder.conf` 的选项“`osapi_volume_extension`”值又分为两种情况：`standard_extensions` 与 `select_extensions`。`standard_extensions` 是指加载 `contrib` 目录下实现的所有资源，`select_extensions` 则可以指定加载哪些资源。`standard_extensions` 为默认的设置。

```
osapi_volume_extension = cinder.api.contrib.standard_extensions
```

如果“`osapi_volume_extension`”选项的值为 `cinder.api.contrib.select_extensions`，则可以在“`osapi_volume_ext_list`”选项中指定要加载资源的列表。

既然 Cinder API 采用了类似 Nova API 的实现方式，则 Cinder API 的执行过程同样类似与 Nova API，从 `cinderclient` 开始算起包括 3 个阶段：`cinderclient` 将用户命令转换为标准 HTTP

请求的阶段；Paste Deploy 将请求路由到具体的 WSGI Application 的阶段，比如 v1 API 对应的 WSGI Application；Routes 将请求路由到具体函数并执行的阶段。

Cinder API 服务 cinder-api 在第二个阶段开始参与，会创建一个 WSGI Server 去监听用户的 HTTP 请求。Paste Deploy 路由的过程主要依赖于配置文件/etc/cinder/api-paste.ini。

```
[composite:osapi_volume]
use = call:cinder.api:root_app_factory
/: apiversions
/v1: openstack_volume_api_v1
/v2: openstack_volume_api_v2
/v3: openstack_volume_api_v3

[composite:openstack_volume_api_v1]
use = call:cinder.api.middleware.auth:pipeline_factory
noauth = request_id faultwrap sizelimit osprofiler noauth apiv1
keystone = request_id faultwrap sizelimit osprofiler authtoken
keystonecontext apiv1
keystone_nolimit = request_id faultwrap sizelimit osprofiler authtoken
keystonecontext apiv1

[composite:openstack_volume_api_v2]
use = call:cinder.api.middleware.auth:pipeline_factory
noauth = request_id faultwrap sizelimit osprofiler noauth apiv2
keystone = request_id faultwrap sizelimit osprofiler authtoken
keystonecontext apiv2
keystone_nolimit = request_id faultwrap sizelimit osprofiler authtoken
keystonecontext apiv2

[composite:openstack_volume_api_v3]
use = call:cinder.api.middleware.auth:pipeline_factory
noauth = cors http_proxy_to_wsgi request_id faultwrap sizelimit osprofiler
noauth apiv3
keystone = cors http_proxy_to_wsgi request_id faultwrap sizelimit
osprofiler authtoken keystonecontext apiv3
keystone_nolimit = cors http_proxy_to_wsgi request_id faultwrap sizelimit
osprofiler authtoken keystonecontext apiv3

[app:apiv1]
paste.app_factory = cinder.api.v1.router:APIRouter.factory

[app:apiv2]
paste.app_factory = cinder.api.v2.router:APIRouter.factory

[app:apiv3]
```

```
paste.app_factory = cinder.api.v3.router:APIRouter.factory
```

有 3 个 WSGI Application, 包括 `apiv1`、`apiv2` 和 `apiv3` 会被加载, 它们对应的分别是 `cinder.api.v1.router.APIRouter`、`cinder.api.v2.router.APIRouter` 和 `cinder.api.v3.router.APIRouter`。这 3 个类继承自 `cinder.api.openstack.APIRouter`。

```
class APIRouter(base_wsgi.Router):
    def __init__(self, ext_mgr=None):
        if ext_mgr is None:
            if self.ExtensionManager:
                ext_mgr = self.ExtensionManager()
            else:
                raise Exception(_("Must specify an ExtensionManager class"))

        mapper = ProjectMapper()
        self.resources = {}
        self._setup_routes(mapper, ext_mgr)
        self._setup_ext_routes(mapper, ext_mgr)
        self._setup_extensions(ext_mgr)
        super(APIRouter, self).__init__(mapper)
```

`APIRouter` 类初始化时, 会调用 `_setup_routes()`、`_setup_ext_routes()`、`_setup_extensions()` 分别建立核心资源与扩展资源的路由信息。具体的资源加载过程与 API 执行过程可以参看第 5 章 Nova API 的介绍。

6.2.3 cinder-scheduler

与 Nova 中的调度服务 `nova-scheduler` 类似, Cinder 的调度服务 `cinder-scheduler` 也用于选择一个合适的节点, 不过不同的是, `nova-scheduler` 选择的是计算节点来响应用户有关虚拟机生命周期相关的请求, 而 `cinder-scheduler` 选择的是 `cinder-volume` 节点来处理用户有关 Volume 生命周期的请求。

同样, `cinder-scheduler` 选择的方式也可以有很多种。为了便于之后的扩展, Cinder 将一个调度器必须要实现的接口提取出来成为 `cinder.scheduler.driver.Scheduler`, 只要继承类 `Scheduler` 并实现其中的接口, 我们就可以实现一个自己的调度器。

目前, Cinder 中只实现了一个调度器 `FilterScheduler`, 但是在历史上曾经存在 `SimpleScheduler` (选择剩余存储空间最多的 Host) 和 `ChanceFilter` (随机挑选满足条件的 Host) 两个调度器, 但是它们现在已经被利用 `FilterScheduler` 的框架重新实现。

不同的调度器不能够共存, 需要在 `/etc/cinder/cinder.conf` 中通过 `scheduler_driver` 选项指定, 默认使用的即是 `FilterScheduler`:

```
scheduler_driver = cinder.scheduler.filter_scheduler.FilterScheduler
```

FilterScheduler 的工作流程基本与 Nova 的 FilterScheduler 调度器相同，如图 6-11 所示。

FilterScheduler 首先使用指定的 Filters（过滤器）得到符合条件的 cinder-volume 节点，比如 Volume 有足够的存储空间，然后对得到的主机列表计算权重并排序，获得最佳的一个。完整来说这个过程可以分为以下几个阶段。

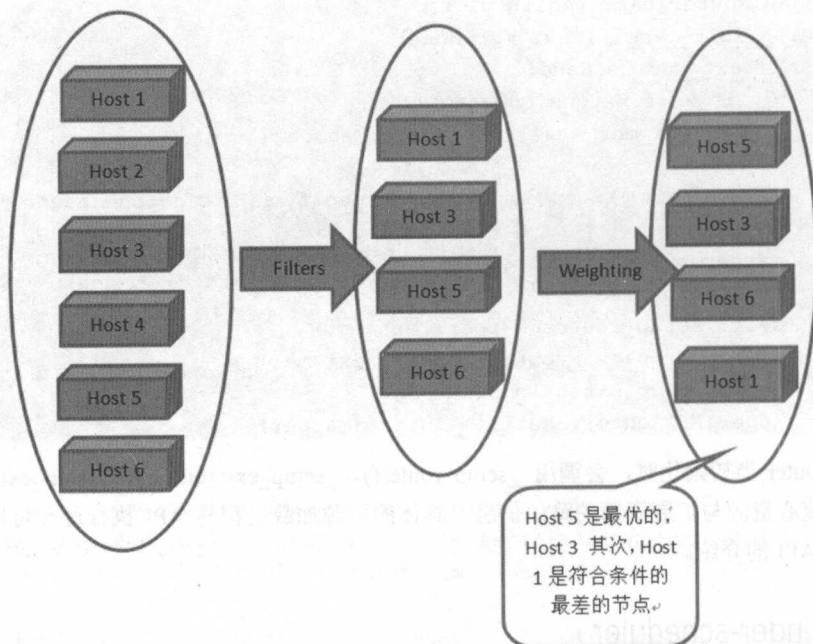


图 6-11 FilterScheduler 工作流程

1. 通过 cinder.scheduler.rpcapi.SchedulerAPI 发出 RPC 请求

通常 OpenStack 项目中各个服务代码所在的目录都会有一个 `rpcapi.py` 文件，其中定义了该服务所能提供的 RPC 接口。对于 `cinder-scheduler` 服务来说，其他服务将 `cinder.scheduler.rpcapi` 模块导入就可以使用其中定义的接口远程调用 `cinder-scheduler` 提供的服务。`cinder-scheduler` 注册的 RPC Server 接收到 RPC 请求后，再由 `cinder.scheduler.manager.SchedulerManager` 真正地选择 `cinder-volume` 节点的操作。

2. 从 SchedulerManager 类到调度器 (Scheduler 类)

`SchedulerManager` 类用于接收 RPC 请求，在一些参数验证之后，将请求交由具体的调度器来处理，它在 RPC 客户端和具体的调度器之间起到一个桥梁的作用。

`SchedulerManager` 类初始化的时候会根据配置文件 `/etc/cinder/cinder.conf` 中的 `scheduler_driver` 的值初始化相应的调度器。

3. Filtering (过滤) 与 Weighting (权重计算与排序)

Filtering 就是使用配置文件指定的各种 Filters 去过滤掉不符合条件的主机, Weighting 则是指对所有符合条件的主机计算权重 (Weight) 并排序, 从而得出最佳的一个。Cinder 中已经实现了几种不同的 Filter 和 Weigher, 所有 Filter 的实现位于 `cinder/scheduler/filters` 目录, 所有 Weigher 的实现位于 `cinder/scheduler/weights` 目录。

Filter 与 Weigher 的实现都有其特定的要求, 比如所有 Filter 都必须继承自类 `cinder.openstack.common.scheduler.filters.BaseHostFilter`, 我们自己也可以很方便地通过继承类 `BaseHostFilter` 来创建一个新的 Filter。新建的 Filter 只需实现一个函数 `host_passes()`, 返回结果只有两种, 满足条件返回 `True`, 否则返回 `False`。

我们可以在配置文件中指定使用哪些 Filter 与 Weigher:

```
scheduler_default_filters=AvailabilityZoneFilter,CapacityFilter,CapabilitiesFilter
scheduler_default_weighters=CapacityWeighter
```

至于更为详细的 Filtering 与 Weighting 处理流程, 与 `nova-scheduler` 类似, 可以参看第 5 章中对 `nova-scheduler` 的介绍。

6.2.4 cinder-volume

类似于 Nova 中虚拟机的生命周期由 `nova-compute` 服务来管理, Cinder 中 Volume 的生命周期则由 `cinder-volume` 服务来管理。

1. cinder-volume 源码目录结构

`cinder-volume` 服务的代码位于 `cinder/volume`。

```
.
├── api.py
├── configuration.py
├── driver.py
├── drivers
├── flows
├── group_types.py
├── manager.py
├── qos_specs.py
├── rpcapi.py
├── targets
├── throttling.py
├── utils.py
└── volume_types.py
```

`rpcapi.py` 文件定义了提供给 RPC 调用的接口 `VolumeAPI`, `api.py` 文件中又对 RPC 的调用

做了一层封装，其他模块需要导入的是 `api` 模块。`manager.py` 是 `cinder-volume` 最为核心的代码，其中的 `VolumeManager` 类用于执行接收到的 RPC 请求，所有有关 `Volume` 生命周期管理的函数都涵盖在内。

如前所述，不同的后端存储技术与存储厂商的设备以 `Driver` 的形式在 `Cinder` 中得以支持，`driver.py` 文件中定义了各种 `Driver` 的基类 `VolumeDriver`，所有具体 `Driver` 的实现都位于 `drivers` 子目录中，`configuration.py` 文件则是为所有的 `Driver` 实现提供一些配置相关的支持。

创建好的 `Volume` 一般通过 `iSCSI Target` 的方式展现给 `Nova`，这样 `Nova` 可以通过 `iSCSI` 协议将其连接到计算节点上供虚拟机使用。`Cinder` 支持多种提供 `iSCSI Target` 的方法，包括 `IET`、`ISER`、`LIO` 以及 `TGT` 等，相关实现位于 `targets` 目录中，默认使用的是 `TGT` (`Linux SCSI Target Framework`)。

`OpenStack` 在 `H` 版中引入了 `QoS` 特性，并在 `Cinder` 中提供了一个 `QoS Spec` (`Quality of Service Specifications`) 框架，每个 `QoS Spec` 与 `Volume Type` 相联系，用户在创建一个卷时可以将该卷与一个 `Volume Type` 联系，这样就间接使得该卷与特定 `QoS Spec` 联系。`qos_specs.py` 中即是 `Qos Spec` 的相关实现，`Volume Type` 的实现位于 `volume_types.py` 文件。

`Cinder` 大量使用了 `TaskFlow` 库来控制任务的执行，`flows` 目录即是相关的实现，其中实现的所有 `Task` 都需要继承类 `cinder.flow_utils.CinderTask`。

2. iSCSI Target

基于 `iSCSI` 能够以较低的门槛实现 `SAN` 的应用。在 `OSI 7` 层模型中，`iSCSI` 属于传输层的协议，规定了 `iSCSI Target` 和 `iSCSI Initiator` 之间的通信机制。`iSCSI Target` 通常是指存储设备，比如存放数据的硬盘或磁盘阵列，`iSCSI Initiator` 则是指能够基于 `iSCSI` 协议访问 `Target` 的客户端软件。

`OpenStack` 中，`Cinder` 创建 `Volume` 之后通常以 `iSCSI Target` 方式提供给 `Nova`，`Nova` 通过 `iSCSI` 协议将该 `Volume` 连接到计算节点上供虚拟机使用，如图 6-12 所示。

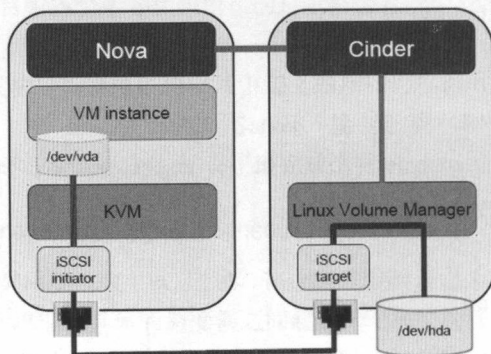


图 6-12 iSCSI Target 方式

3. 后端存储 Driver

为了支持不同的后端存储技术与设备，Cinder 创建了一个 Driver 框架，将所有 Driver 需要实现的接口包含在 `cinder.volume.driver.VolumeDriver` 类中，我们可以在 `cinder` 配置文件中指定使用哪种后端存储的 Driver，且以哪种方式提供 iSCSI Target，Cinder 默认使用的是 `LVMISCSIDriver`：

```
volume_driver = cinder.volume.drivers.lvm.LVMISCSIDriver
iscsi_helper = tgtadm
```

`cinder.volume.manager.VolumeManager` 类在初始化时会根据配置文件的设置初始化指定的 Driver，以默认的 `LVMISCSIDriver` 为例，上述 `VolumeManager`、`VolumeDriver` 与具体 iSCSI Target 提供方式的关系如图 6-13 所示。

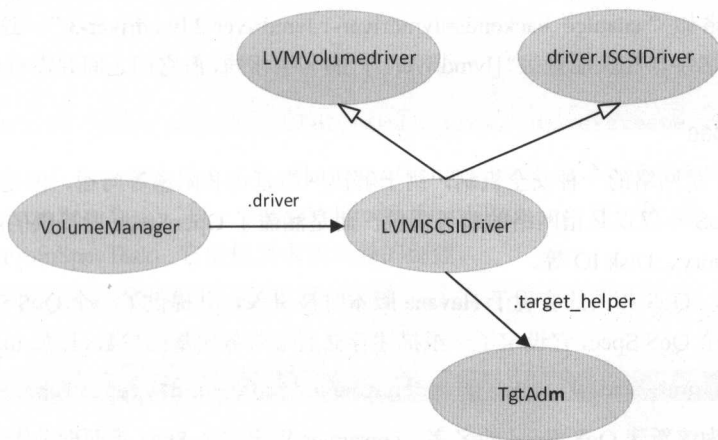


图 6-13 VolumeManager、VolumeDriver 与具体 iSCSI Target 提供方式的关系

4. Volume Type

Cinder 可以支持多个或多种存储后端（Multiple-Storage Backends）并存，每个存储后端都有自己的名字，但是这个名字并不要求是唯一的，可以被共用，此时 `cinder-scheduler` 会根据 Filter 来选择在哪个存储后端上创建 volume。

存储后端的名称通过 Volume Type 的 `extra-specs` 来设置。Volume Type 是卷的一种标识，我们可以自由地创建和删除。Cinder 中与 Volume Type 相关的资源，或者说 API 有两种：`type` 和 `extra_specs`。针对 `type` 的操作有创建、删除、查询等；针对 `extra_specs` 的操作主要就是 `set` 与 `unset`，`set` 是传入一个 `key/value` 对，`unset` 只需传入一个 `key` 值，表示删除与这个 `key` 值匹配的 `extra_spec`。

存储后端的名称就是通过指定 `volume_backend_name` 的键值来进行设置，比如：

```
$ cinder type-create lvm
```

```
$ cinder type-list
$ cinder type-key lvm set volume_backend_name=LVM_iSCSI
$ cinder extra-specs-list
```

上述命令创建了一个存储后端类型“lvm”，并且指定它的名字为“LVM_iSCSI”。每个存储后端在配置文件中都有一组相关的配置，比如使用 DevStack 部署时，默认如下：

```
default_volume_type = lvmdriver-1
enabled_backends = lvmdriver-1
[lvmdriver-1]
volume_group = stack-volumes-lvmdriver-1
volume_driver = cinder.volume.drivers.lvm.LVMVolumeDriver
volume_backend_name = lvmdriver-1
```

我们必须设置 `enabled_backends` 选项来指定使用的存储后端，如果有多个，则需要使用“,” 隔开，比如“`enabled_backends=lvmdriver-1,lvmdriver-2,lvmdriver-3`”。这里存储后端 `lvmdriver-1` 的名字与相关配置组“`[lvmdriver-1]`”的名字相同，但它们之间并没有必然的联系。

5. QoS Spec

通常 QoS 是网络的一种安全机制，用于解决网络延迟和阻塞等问题，但是在 OpenStack 中，这里的 QoS 不仅仅是指网络的服务质量，而是涵盖了 OpenStack 所提供的所有资源，包括 CPU、Memory、Disk IO 等。

Cinder 中与 QoS 相关的实现于 Havana 版本时被引入，并提供了一个 QoS Spec 框架，我们可以创建一个 QoS Spec，它设定了一组描述存储后端服务质量的参数，比如 `total_bytes_sec`。

```
$ cinder qos-create read_qos consumer="front-end" read_iops_sec=1000
```

`read_qos` 为该新建 QoS Spec 的名字，`consumer` 指定这个 Spec 是面向前端（Hypervisor）还是存储后端。

Cinder 中一个卷只能与 Volume Type 相关联，而不能与 QoS Spec 关联，所以为了将卷与 QoS Spec 关联，我们必须首先创建一个 Volume Type，然后将其与 QoS Spec 进行关联：

```
$ cinder qos-associate [qos-spec-id] [type-id]
```

然后在创建一个卷时将该卷与 Volume Type 关联，这样就间接使得该卷与特定的 QoS Spec 关联起来，此后在该卷附加（Attach）到一个虚拟机上时，即可实现该虚拟机的限速。如果存储后端本身就支持 QoS Spec 中的设定，比如速度限制，那么也可以通过将 `consumer` 指定为“back-end”来通过存储后端实现。

6. Volume 创建过程

对于一个 Volume 的创建过程，从 `cinderclient` 到具体 API 执行函数 `cinder.api.v3.volumes.VolumeController.create()` 的过程已经在 Cinder API 部分进行了介绍，这里的内容只涉及后续的操作。

cinder.api.v3.volumes.VolumeController.create()中会通过 RPC 远程调用 cinder-volume 服务的 cinder.volume.manager.VolumeManager.create_volume() 函数来完成具体的创建。create_volume()函数的主要工作就是利用 TaskFlow 库构建创建卷的 flow 并执行。

```
$ cinder/volume/flows/manager/create_volume.py

def get_flow(context, manager, db, driver, scheduler_rpcapi, host, volume,
              allow_reschedule, reschedule_context, request_spec,
              filter_properties, image_volume_cache=None):
    volume_flow.add(ExtractVolumeSpecTask(db),
                    NotifyVolumeActionTask(db, "create.start"),
                    CreateVolumeFromSpecTask(manager,
                                              db,
                                              driver,
                                              image_volume_cache),
                    CreateVolumeOnFinishTask(db, "create.end"))

    return taskflow.engines.load(volume_flow, store=create_what)
```

创建 Volume 的 flow 共添加了 4 个 task: ExtractVolumeSpecTask、NotifyVolumeActionTask、CreateVolumeFromSpecTask 和 CreateVolumeOnFinishTask。其中最重要的为 CreateVolumeFromSpecTask，它根据要求实现卷的创建。

```
$ cinder/volume/flows/manager/create_volume.py

class CreateVolumeFromSpecTask(flow_utils.CinderTask):
    def execute(self, context, volume_ref, volume_spec):
        if create_type == 'raw':
            model_update = self._create_raw_volume(volume, **volume_spec)
        elif create_type == 'snap':
            model_update = self._create_from_snapshot(context, volume,
                                                      **volume_spec)
        elif create_type == 'source_vol':
            model_update = self._create_from_source_volume(
                context, volume, **volume_spec)
        elif create_type == 'source_replica':
            model_update = self._create_from_source_replica(
                context, volume, **volume_spec)
        elif create_type == 'image':
            model_update = self._create_from_image(context,
                                                    volume,
                                                    **volume_spec)
```

CreateVolumeFromSpecTask 区分了 5 种创建 Volume 的方式：建立 raw 格式的新卷、从快照建立新卷、从已有的卷建立新卷、从副本建立新卷和从镜像建立新卷。对于建立 raw 格式

新卷的情况，将直接调用指定 Driver 的 `create_volume()` 函数进行创建。对于默认的 `LVMVolumeDriver`，就是直接调用“`lvcreate`”命令创建 Volume。

至此，一个新的 Volume 被创建成功，此后，Nova 会根据 Volume 的 ID 调用 `cinder.volume.manager.VolumeManager.initialize_connection()`。该函数会根据指定的方式（比如默认的 TGT）创建 iSCSI Target，并返回该 Target 的相关信息，比如 iSCSI 的 `iqn` (iSCSI Qualified Name)，之后 Nova 就可以通过该存储节点的 IP 地址和 `iqn` 来连接并且挂载这个 Volume。

6.2.5 cinder-backup

`cinder-backup` 用于将 Volume 备份到其他存储系统上。目前支持的备份存储系统有 Swift、Ceph、IBM Tivoli Storage Manager (TSM)、GlusterFS 等，默认为 Swift。

`cinder-backup` 服务的代码位于 `cinder/backup`：

```
.
├── api.py
├── chunkedriver.py
├── driver.py
├── drivers
│   ├── ceph.py
│   ├── glusterfs.py
│   ├── google.py
│   ├── nfs.py
│   ├── posix.py
│   ├── swift.py
│   └── tsm.py
├── manager.py
└── rpcapi.py
```

类似于 `cinder-volume` 服务，`rpcapi.py` 文件定义了提供给 RPC 调用的接口 `BackupAPI`，`api.py` 文件中又对 RPC 的调用做了一层封装，其他模块需要导入的是 `api` 模块。`manager.py` 是 `cinder-backup` 最为核心的代码，其中的 `BackupManager` 类用于执行接收到的 RPC 请求。

不同的备份存储系统以 Driver 的形式得以支持，`driver.py` 文件中定义了各种 Driver 的基类 `BackupDriver`，所有具体 Driver 的实现都位于 `drivers` 子目录，通过配置文件的 `backup_driver` 选项指定使用的 Driver。

```
backup_driver = cinder.backup.drivers.swift
```

当前 Cinder 只支持设置一个备份存储后端。从 Mitaka 开始，`backup` 服务和 `volume` 服务解除了紧耦合，不再需要装在同一台主机上。`cinder-backup` 服务在接到请求后任何挑选一个 `backup host` 来提供备份服务。

- 创建备份：`cinder-backup` 通过 RPC 请求 `cinder-volume` 服务提供需要备份的卷（`get_backup_device`）。如果需要备份的卷处于 `available` 状态，直接把该卷返回给

cinder-backup。如果需要备份的卷正在被使用，则先根据该卷创建一份快照或者克隆卷，返回快照或者克隆卷给 cinder-backup。cinder-backup 收到备份卷后，把备份卷挂载到本机，将数据备份到后端备份存储，如图 6-14 所示。

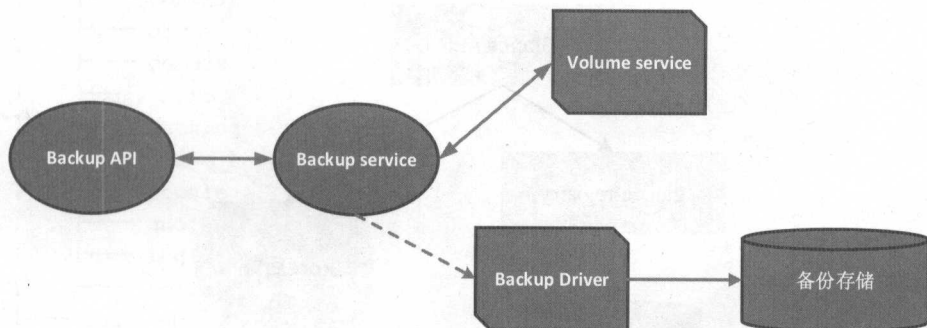


图 6-14 cinder-backup 工作流程

- 恢复备份：cinder-backup 将需要进行数据还原的卷挂载到本机，将数据从备份存储读出，恢复到卷上。
- 删除备份：cinder-backup 直接调用 Backup Driver 中的接口进行删除。

6.3 Glance

Glance 为 OpenStack 提供虚拟机的镜像服务。作为与 Swift、Cinder 并列的存储相关的三驾马车之一，Glance 本身却并不负责大量数据的存储，它对镜像的存储需要依赖于 Swift 等项目来完成。

6.3.1 Glance 体系结构

由于 Glance 并不负责实际的存储，只是完成一些镜像管理的工作，因此它的功能比较单一，包含的主要组件也相对比较少。图 6-15 所示为 Glance 的体系结构。

由图 6-15 可以看出，Glance 主要由 glance-api 与 glance-registry 两个服务组成。glance-api 是进入 Glance 的入口，负责接收用户的 RESTful 请求，然后通过后台的 Swift、Amazon S3 等存储系统完成镜像的存储与获取。

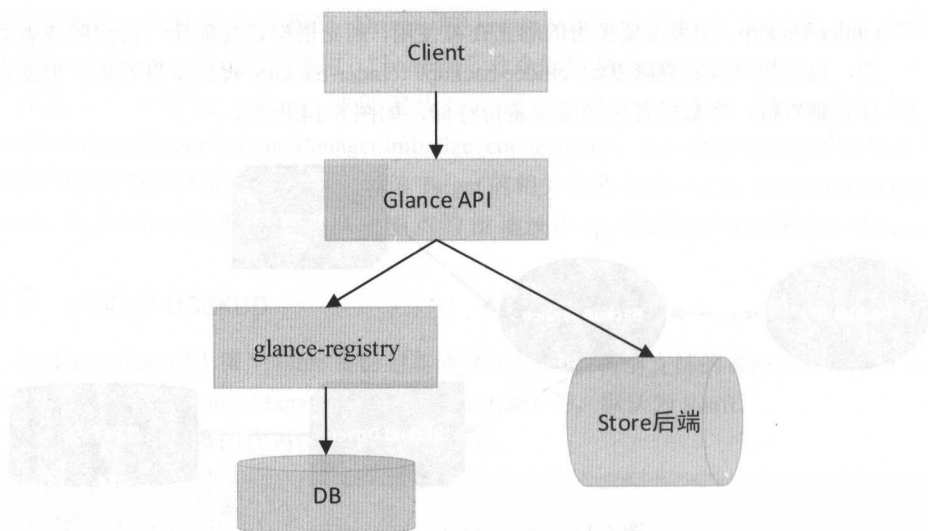


图 6-15 Glance 体系结构

Glance 的 Store 模块实现了一个存储后台的框架，根据这个框架所提供的接口，实现了对各种不同后台存储系统的支持，包括 Amazon 的 S3、Cinder/Swift，还有诸如 Ceph、Sheepdog、GlusterFS 等分布式存储。在 Juno 版本之前，Store 模块的实现位于 glance/store 目录；Juno 版本之后，Store 模块作为一个独立的项目 glance_store 被剥离出来 (https://github.com/openstack/glance_store)，以便为更多的项目服务，如 Nova 等，但事实上目前为止，并没有 Glance 之外的其他项目使用到 glance_store。

与 glance-api 服务一样，glance-registry 也是一个 WSGI Server，不过不同的是，glance-registry 处理的是与镜像元数据相关的 RESTful 请求。glance-api 接收到用户的 RESTful 请求后，如果该请求与元数据相关，则将其转发给 glance-registry 服务。

然后 glance-registry 会解析请求的内容，并与数据库进行交互，存取或更新镜像的元数据，这里的元数据是指保存在数据库中的关于镜像的一些信息，Glance 的 DB 模块存储的仅仅是镜像的元数据。

不过上述的 Glance 结构仅仅针对于 Glance API 的 v1 版本，在 v2 API 版本时，glance-registry 服务的内容被整合进了 glance-api 之中。如果 glance-api 接收到与镜像元数据有关的请求，会直接操作数据库，不需要再通过 glance-registry 服务。

1. Glance 源码目录结构

```

.
├── api-ref
├── etc
└── glance
  
```

```
|   |—— api
|   |—— async
|   |—— cmd
|   |—— common
|   |—— contrib
|   |—— db
|   |—— domain
|   |—— glare
|   |—— hacking
|   |—— image_cache
|   |—— locale
|   |—— quota
|   |—— registry
|   |—— tests
|—— rally-jobs
|—— setup.cfg
|—— setup.py
```

Glance 的核心代码位于 `glance` 目录，所有服务以及工具的执行脚本位于 `glance/cmd`，`glance/api`、`glance/registry` 分别对应 Glance API 与 `glance-registry` 服务的具体实现。

`glance-api` 可以为镜像建立本地缓存，实现 API 服务节点的数量扩展，提供多个 API 节点为同一个镜像提供服务的效率。如果只有一个 API 节点，缓存机制并没有意义。本地的 Image Cache 是镜像文件的完全复制，这种缓存机制对用户来说是透明的，用户并不知道得到的镜像文件是来自于存储后台还是本地的缓存。

用户可以通过配置文件 `etc/glance/glance-cache.conf` 指定 Cache 文件存放的路径、本地能够用于 Cache 的存储空间等信息。镜像缓存的实现位于 `glance/image_cache` 目录。

针对 `import`、`export`、`clone` 等镜像操作，Glance 统一引入了 Task 的概念，从而方便管理。Task 是针对镜像的异步操作，`glance/async` 即是部分实现。

Glance 采用了责任链（Chain of Responsibility）的设计模式实现用户请求的处理流程。在责任链模式里，各个对象间通过前一个对象对后一个对象引用而连接起来形成一条链，请求在这个链上传递，直到链上的某一个对象决定处理此请求。发起请求的用户或客户端并不知道链上的哪一个对象最终处理了这个请求，从而使系统可以在不影响客户端的情况下动态地重新组织链和分配责任。`glance/domain` 目录与 `glance/gateway.py` 文件即是相关的一些实现，`glance.domain` 模块定义了一些基类或接口，比如 `ImageFactory`、`Repo` 等，而 `glance.gateway.Gateway` 模块则是实现了责任链的建立。

Rally 是一个用于性能测试的项目，`glance/rally-jobs` 目录是一些用于 Rally 的文件或插件。

2. setup.cfg

依照惯例，分析具体的实现之前，我们首先浏览 `setup.cfg` 文件。

```

console_scripts =
    glance-api = glance.cmd.api:main
    glance-cache-prefetcher = glance.cmd.cache_prefetcher:main
    glance-cache-pruner = glance.cmd.cache_pruner:main
    glance-cache-manage = glance.cmd.cache_manage:main
    glance-cache-cleaner = glance.cmd.cache_cleaner:main
    glance-control = glance.cmd.control:main
    glance-manage = glance.cmd.manage:main
    glance-registry = glance.cmd.registry:main
    glance-replicator = glance.cmd.replicator:main
    glance-scrubber = glance.cmd.scrubber:main
    glance-glare = glance.cmd.glare:main

```

“entry_points” 中的命名空间 “console_scripts” 里，涵盖了 Glance 所提供的所有服务以及工具，其中的每一项都表示一个可执行的脚本，这些脚本在部署时会被安装，它们同时也是 Glance 各项工作的入口。

- **glance-cache-***: 4 个对 Image Cache 进行管理的工具。比如 **glance-cache-pruner** 用于执行一些周期性的任务，**glance-cache-cleaner** 可以清理 Cache 文件释放空间。
- **glance-manage**: 用于 Glance 数据库的管理。
- **glance-replicator**: 用于实现镜像的复制。
- **glance-scrubber**: 用于清理已经删除的 Image。
- **glance-control**: Glance 提供了 **glance-api** 和 **glance-registry** 两个 WSGI Server，以及一个 **glance-scrubber** 后台服务进程，这里的 **glance-control** 工具即是用于控制这 3 个服务进程，包括 **start**、**stop**、**restart** 等。
- **glance-glare**: Glare (Glance Artifact Repository) API 服务，目前正在开发中。

6.3.2 Glance API

Glance API 主要提供镜像的管理功能，比如 Image 的导入/导出、镜像元数据的管理等。目前 GlanceAPI 共有 v1 和 v2 两个版本，v2 版本整合了 **glance-registry** 服务的功能，并且采用责任链设计模式来实现了 API 的处理流程。具体 API 的格式可查询 <http://docs.openstack.org/developer/glance/glanceapi.html>。

Glance API 的执行从 **glanceclient** 发送 HTTP 请求，**glance-api** 服务接收请求并处理的整个过程与 **Cinder** 等其他项目基本相同，**glance-api** 与 **glance-registry** 分别有自己的 Paste Deploy 配置文件 **/etc/glance/glance-api-paste.ini** 与 **/etc/glance/glance-registry-paste.ini**，可参考前面章节其他项目的介绍。这里需要提及的是，**glance-api** 服务 (**glance/cmd/api.py**) 在初始时，会导入前面所说的 **glance_store** 项目，并初始化后台的存储系统。

1. Image

Image 是 Glance 所管理的主要资源。类似于 VMware 的 VM 模板 (Template)，它预先安装了 OS。如果从 Image 启动 VM，该 VM 被删除后，Image 依然存在，但是 Image 上不包含本次在该 VM 实例上的修改，因为 Image 只是给 VM 启动的模板。

相对于整个 OpenStack，或者说 Nova 是一个虚拟机的世界。虚拟机是这个世界的主题，Glance 则是一个主体为 Image 的小世界，能够准确完整地去描述一个 Image 必然也是 Glance 的重点，比如：

- id, 唯一标识 Image 的 UUID。
- name, Image 名字。
- owner, Image 的拥有者。
- size, 字节表示的 Image 大小。
- created_at、updated_at 等，表示 Image 的“出生时间”、最后一次被修改的时间等。
- location, Image 存储的位置，如果是普通文件系统的方式，则类似于 file:///var/lib/glance/images/12809466-fffd-4bdf-9227-c3c05c28a2c5；如果是 S3、Swift 等其他后台存储系统，则为形如 s3://<ACCESS_KEY>:<SECRET_KEY>@<S3_URL>/<BUCKET>/<OBJ>的 URL。
- disk_format, 磁盘格式，也可以理解为 Image 本身的格式。比如 raw、qcow2（用于 Qemu）、vdi（用于 Virtual Box）、vmdk（用于 VMware）等。
- status, 镜像的状态。类似于 Nova 负责管理虚拟机的生命周期，Cinder 负责管理 Volume 的生命周期，Glance 则负责管理 Image 的生命周期。既然是存在生命周期，Image 必然存在各种状态以及状态之间的演化，如图 6-16 所示。
- queued, 表明镜像 ID 已经被保留，但是镜像数据还没有上传。
- saving, 表明此时镜像正在上传。
- active, 是 Image 成功上传完毕后的状态，此时该 Image 完全可用。
- killed, 表明上传时发生错误，此时 Image 完全不可用。“killed”在 v2 中被废除，如果上传失败，状态将转变为“queued”以便上传可以重试。
- deleted, 虽然此时 Glance 还保留 Image 的相关信息，但是该 Image 已不可用，在未来某个时候会被 glance-scrubber 彻底删除。
- pending_delete, 和 deleted 类似，但是并不会删除 Image，此时尚可恢复。

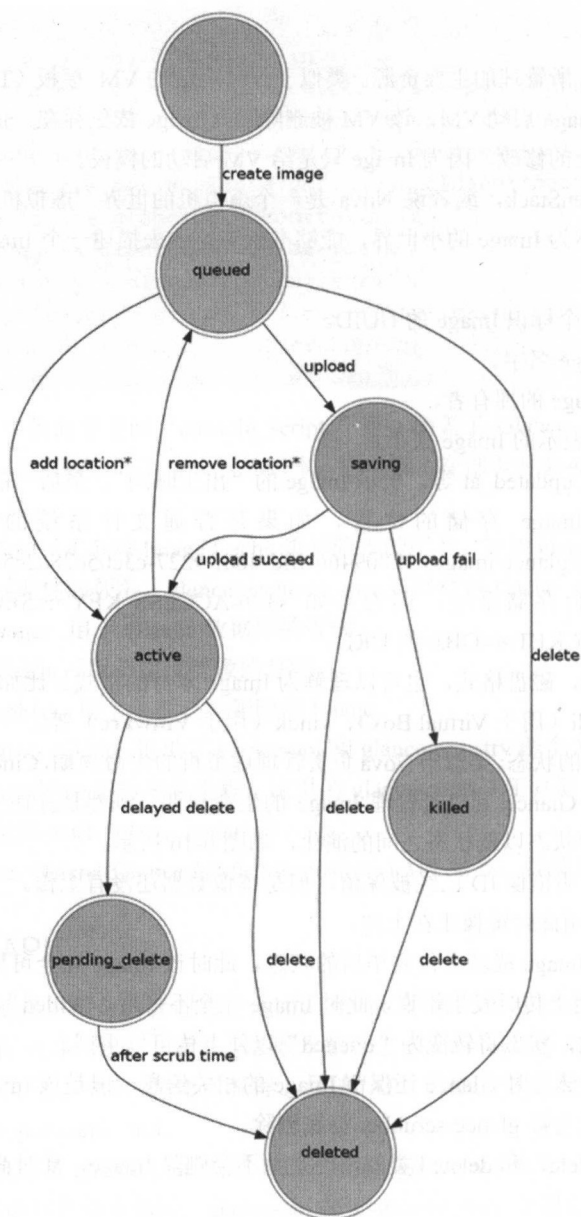


图 6-16 Image 状态演变

2. Task

一般来说，对 Image 的操作有 import、export、clone 等几种。Glance 把这些操作统一起来抽象出了 Task 的概念来方便管理。Task 是针对 Image 的异步（Async）操作，具有的一些

属性包括 id、owner、状态等。Glance 同时也实现了统一的 JSON 格式的 API 来操作这些 Task，比如创建、删除、查询状态等。

在一个 Task 运行过程中，我们可以不断查询它的状态。Task 的状态有以下 4 种：

- pending，表示 Task 被创建，但并未执行。
- processing，表示 Task 正在执行中。
- success，表示 Task 成功结束。
- failure，表示 Task 由于某种原因未能成功结束。

Task 和 Image 的操作完全是两个概念：首先它们都是不同的 API 资源；其次，Task 是异步的操作，是对 Image 操作的封装，目前只对 clone、import、export 3 种操作进行了封装；最后，Task 一旦创建，可以不断查询它的状态，在一次操作比如 import 之后，Task 可以消亡，但是此时生成的 Image 依然存在。

3. Image 创建过程

glance-api 接收到用户 HTTP 请求后，会通过 WSGI Routes 模块路由到具体的操作函数，我们可以看到对 v2 版本的 API 来说，很多操作函数的开始都会有形如 “*_factory = self.gateway.get_*()” 与 “*_repo = self.gateway.get_*()” 的语句，对于 Image 的创建则是：

```
$ glance/api/v2/images.py

class ImagesController(object):

    def create(self, req, image, extra_properties, tags):
        image_factory = self.gateway.get_image_factory(req.context)
        image_repo = self.gateway.get_repo(req.context)
        try:
            image = image_factory.new_image(extra_properties=extra_properties,
                                           tags=tags, **image)
            image_repo.add(image)
        .....

    return image
```

首先会使用 glance.gateway.Gateway 模块获取两个对象：image_factory 和 image_repo，其中，image_factory 对应针对后端存储系统进行镜像存取等操作，image_repo 完成针对 Glance 数据库的管理。

image_factory 和 image_repo 是 glance.gateway.Gateway 模块利用责任链设计模式建立的两条完成请求处理的责任链。

```
$ glance/gateway.py

class Gateway(object):
```

```

def get_image_factory(self, context):
    image_factory = glance.domain.ImageFactory()
    store_image_factory = glance.location.ImageFactoryProxy(
        image_factory, context, self.store_api, self.store_utils)
    quota_image_factory = glance.quota.ImageFactoryProxy(
        store_image_factory, context, self.db_api, self.store_utils)
    policy_image_factory = policy.ImageFactoryProxy(
        quota_image_factory, context, self.policy)
    notifier_image_factory = glance.notifier.ImageFactoryProxy(
        policy_image_factory, context, self.notifier)
    if property_utils.is_property_protection_enabled():
        property_rules = property_utils.PropertyRules(self.policy)
        pif = property_protections.ProtectedImageFactoryProxy(
            notifier_image_factory, context, property_rules)
        authorized_image_factory = authorization.ImageFactoryProxy(
            pif, context)
    else:
        authorized_image_factory = authorization.ImageFactoryProxy(
            notifier_image_factory, context)
    return authorized_image_factory

def get_repo(self, context):
    image_repo = glance.db.ImageRepo(context, self.db_api)
    store_image_repo = glance.location.ImageRepoProxy(
        image_repo, context, self.store_api, self.store_utils)
    quota_image_repo = glance.quota.ImageRepoProxy(
        store_image_repo, context, self.db_api, self.store_utils)
    policy_image_repo = policy.ImageRepoProxy(
        quota_image_repo, context, self.policy)
    notifier_image_repo = glance.notifier.ImageRepoProxy(
        policy_image_repo, context, self.notifier)
    if property_utils.is_property_protection_enabled():
        property_rules = property_utils.PropertyRules(self.policy)
        pir = property_protections.ProtectedImageRepoProxy(
            notifier_image_repo, context, property_rules)
        authorized_image_repo = authorization.ImageRepoProxy(
            pir, context)
    else:
        authorized_image_repo = authorization.ImageRepoProxy(
            notifier_image_repo, context)

    return authorized_image_repo

```

如图 6-17 所示的片段,“new_image()”与“add()”分别在责任链 image_factory 和 image_repo

上进行传递，如果这个操作在链上某个类中（比如 glance.location.ImageFactoryProxy）有对应的同名方法，则调用这个方法然后再将请求传递到下一个类，否则就直接传递至下一个。

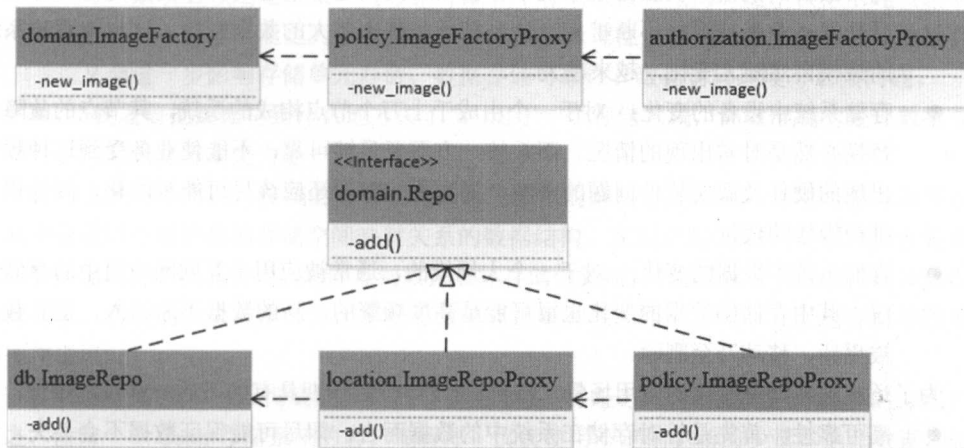


图 6-17 Image 创建过程

6.4 Ceph

随着云计算的发展，传统的存储设备产品越来越显示出局限性，高昂的价格以及难以扩展的架构，使它难以满足很多用户的需求，而 Ceph 的创建正是为了能够改善这种状况。

Ceph 起源于 Sage Weil 在 2004 年的一片博士学术论文，最初是一项关于存储系统的 PhD 研究项目，到了 2010 年 3 月底，2.6.34 Linux 内核也开始对 Ceph 进行支持。

Ceph 遵循 LGPL 协议。作为一个典型的强调性能的系统项目，Ceph 使用 C++ 语言开发。

Ceph 从最初发布到逐渐流行经历了多年的时间，近年来来自 OpenStack 社区的实际需求又让其热度骤升。目前 Ceph 已经成为 OpenStack 社区中呼声最高的开源存储方案之一。

Ceph 的官方定义如下：

“Ceph is a unified, distributed storage system designed for excellent performance, reliability and scalability.”

Ceph 是一种为优秀的性能、可靠性和可扩展性而设计的统一的、分布式的存储系统。

这里面比较关键的两个词是“unified”统一的和“distributed”分布式的。“统一的”意味着 Ceph 可以以一套存储系统同时提供对象存储、块存储和文件系统存储 3 种功能，以便在满足不同应用需求的前提下简化部署和运维。“分布式的”则意味着无中心结构，系统规模可以没有理论上限的扩展。

Ceph 最初针对的目标应用场景，就是大规模的、分布式的存储系统。在 Sage 的思想中，Ceph 需要很好地适应这样一个大规模存储系统的动态特性（以下内容来自章宇的《Ceph 浅析》），

网址: <http://www.tuicool.com/articles/Z7Bb6nb>)。

- 存储系统规模的变化: 这样大规模的存储系统, 往往不是在建设的第一天就能预料到其最终的规模, 甚至是根本就不存在最终规模这个概念的。只能是随着业务的不断开展, 业务规模的不断扩大, 让系统承载越来越大的数据容量。这也就意味系统的规模自然随之变化, 越来越大。
- 存储系统中设备的变化: 对于一个由成千上万个节点构成的系统, 其节点的故障与替换必然是时常出现的情况。而系统一方面要足够可靠, 不能使业务受到这种频繁出现的硬件及底层软件问题的影响, 同时另一方面还应该尽可能智能化, 降低相关维护操作的代价。
- 存储系统中数据的变化: 对于一个大规模的, 通常被应用于互联网应用中的存储系统, 其中存储的数据的变化也很可能是高度频繁的。新的数据不断写入, 已有数据被更新、移动乃至删除。

为了适应这样动态变化的应用场景, Ceph 在设计时就预期具有如下的一些技术特性:

- 高可靠性。首先是针对存储在系统中的数据而言, 即尽可能保证数据不会丢失。其次, 也包括数据写入过程中的可靠性, 即在用户将数据写入 Ceph 存储系统的过程中, 不会因为意外情况的出现造成数据丢失。
- 高度自动化。具体包括了数据的自动 replication、自动 re-balancing、自动 failure detection 和自动 failure recovery。总体而言, 这些自动化特性一方面保证了系统的高度可靠, 另一方面也保障了在系统规模扩大之后, 其运维难度仍能保持在一个相对较低的水平。
- 高可扩展性。这里的“可扩展”概念比较广义, 既包括系统规模和存储容量的可扩展, 也包括随着系统节点数增加的聚合数据访问带宽的线性扩展, 还包括基于功能丰富强大的底层 API 提供多种功能、支持多种应用的功能性可扩展。

针对上述技术特性, Sage 对于 Ceph 的设计思路基本上可以概括为以下两点:

- 充分发挥存储设备自身的计算能力。事实上, 采用具有计算能力的设备(最简单的例子就是普通的服务器)作为存储系统的存储节点, 这种思路即便在 Ceph 发布的当时来看也并不新鲜。但是, Sage 认为那些已有系统基本上都只是将这些节点作为功能简单的存储节点。而如果充分发挥节点上的计算能力, 则可以实现上述预期的技术特性。这一点成为了 Ceph 系统设计的核心思想。
- 去除所有的中心点。一旦系统中出现中心点, 则一方面引入单点故障点, 另一方面也必然面临当系统规模扩大时的规模和性能瓶颈。除此之外, 如果中心点出现在数据访问的关键路径上, 事实上也必然导致数据访问的延迟增大。而这些显然都是 Sage 所设想的系统中不应该出现的问题。虽然在大多数系统的工程实践中, 单点故障点和性能瓶颈的问题可以通过为中心点增加备份加以缓解, 但 Ceph 系统最终采用创新的方法更为彻底地解决了这个问题。

一般而言，一个大规模分布式存储系统，必须要能够解决两个最基本的问题：

- “我应该把数据写入到什么地方”。对于一个存储系统，当用户提交需要写入的数据时，系统必须迅速决策，为数据分配一个存储位置和空间。这个决策的速度影响到数据写入延迟，而更为重要的是，其决策的合理性也影响着数据分布的均匀性。这又会进一步影响存储单元寿命、数据存储可靠性、数据访问速度等后续问题。
- “我之前把数据写到什么地方去了”。对于一个存储系统，高效准确的处理数据寻址问题也是基本能力之一。

针对上述两个问题，传统的分布式存储系统常用的解决方案是引入专用的服务器节点，在其中存储用于维护数据存储空间映射关系的数据结构。在用户写入/访问数据时，首先连接这一服务器进行查找操作，待决定/查到数据实际存储位置后，再连接对应节点进行后续操作。由此可见，传统的解决方案一方面容易导致单点故障和性能瓶颈，另一方面也容易导致更长的操作延迟。

针对这一问题，Ceph 彻底放弃了基于查表的数据寻址方式，而改用基于计算的方式。简言之，任何一个 Ceph 存储系统的客户端程序，仅仅使用不定期更新的少量本地元数据，加以简单计算，就可以根据一个数据的 ID 决定其存储位置。对比之后可以看出，这种方式使得传统解决方案的问题一扫而空。Ceph 的几乎所有优秀特性都是基于这种数据寻址方式实现的。

从软件工程的角度，当我们拿到一份系统需求，明白它所解决的问题以及预期拥有的技术特性和质量需求并进行架构设计时，主要完成的工作有 3 个：一是勾勒它的概念空间，所谓的概念空间就是将要引入的一些核心的概念，比如提到操作系统，我们就会想到进程、进程调度、系统调用等；二是分层，即从逻辑、物理、通用性等角度划分它的层次，比如一个视频监控系统从物理上可以划分为监控端、客户端、平台层；三是模块划分，将之前得到的层次进行细化，或者在每个层次内部引入粒度更小的分区，或者将一些通用的机制进行提取，通过种种这样的手段将系统细化为不同的模块。

接下来我们从这 3 个角度对 Ceph 进行探究。

6.4.1 Ceph 体系结构

首先作为一个存储系统，Ceph 在物理上必然包含一个存储集群，以及一些访问这个存储集群的应用或客户端。Ceph 客户端又需要一定的协议与 Ceph 存储集群交互，因此 Ceph 的逻辑层次演化如图 6-18 所示。

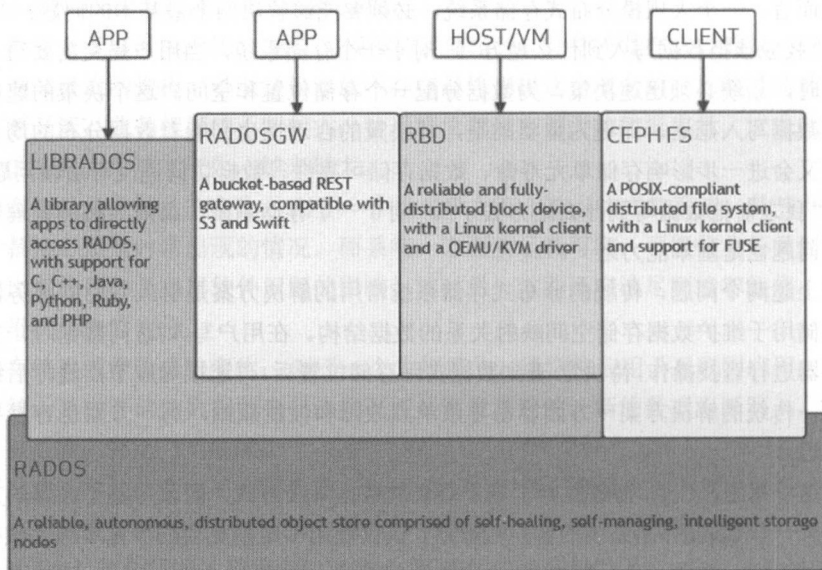


图 6-18 Ceph 逻辑层次

(1) Ceph 存储集群

Ceph 基于 RADOS (A reliable, autonomous, distributed object storage) 提供了一个无限可扩展的存储集群。RADOS 即“可靠的、自动化的、分布式的对象存储”，顾名思义，这一层本身就是一个完整的对象存储系统，所有存储在 Ceph 系统中的用户数据事实上最终都是由这一层来存储的。而 Ceph 的高可靠、高可扩展、高性能、高自动化等特性本质上也是由这一层所提供的。因此，理解 RADOS 是理解 Ceph 的基础与关键。

物理上，RADOS 由大量的存储设备节点组成，每个节点拥有自己的硬件资源（如 CPU、内存、硬盘、网络），并运行着操作系统和文件系统。

(2) 基础库 librados

Ceph 客户端用一定的协议和存储集群交互，Ceph 把此功能封装进了 librados 库，这样基于 librados 库我们就能创建自己的定制客户端。

librados 实际上是对 RADOS 进行抽象和封装，并向上层提供 API，以便可以基于 RADOS（而不是整个 Ceph）进行应用开发。特别要注意的是，RADOS 是一个对象存储系统，因此，librados 实现的 API 也只是针对对象存储功能的。

RADOS 采用 C++ 开发，所提供的原生 librados API 包括 C 和 C++ 两种。物理上，librados 和基于其上开发的应用位于同一台机器，因而也被称为本地 API。应用调用本机上的 librados API，再由后者通过 socket 与 RADOS 集群中的节点通信并完成各种操作。

(3) 高层应用接口 RADOS GW、RBD 与 Ceph FS

这一层的作用是在 librados 库的基础上提供抽象层次更高、更便于应用或客户端使用的上

层接口。

Ceph 对象网关 RADOS GW (RADOS Gateway) 是一个构建在 librados 之上的对象存储接口, 为应用访问 Ceph 集群提供了一个与 Amazon S3 和 Swift 兼容的 RESTful 风格的 gateway。

RBD (Reliable Block Device) 则提供了一个标准的块设备接口, 常用于在虚拟化的场景下为虚拟机创建 volume。Red Hat 已经将 RBD 驱动集成在 KVM/QEMU 中, 以提高虚拟机访问性能。

Ceph FS 是一个 POSIX 兼容的分布式文件系统, 使用 Ceph 存储集群来存储数据。

(4) 应用层

这一层就是不同场景下对于 Ceph 各个应用接口的各种应用方式, 例如基于 librados 直接开发的对象存储应用, 基于 RADOS GW 开发的对象存储应用, 基于 RBD 实现的云硬盘等。

6.4.2 RADOS

如图 6-19 所示, RADOS 集群主要由两种节点组成。一种是为数众多的、负责完成数据存储和维护功能的 OSD (Object Storage Device), 另一种则是若干个负责完成系统状态检测和维持的 Monitor。OSD 和 Monitor 之间相互传输节点状态信息, 共同得出系统的总体工作状态, 并形成全局系统状态记录数据结构, 即所谓的集群运行图 (Cluster Map)。集群运行图与 RADOS 提供的特定算法相配合, 便实现了 Ceph 的诸多优秀特性。

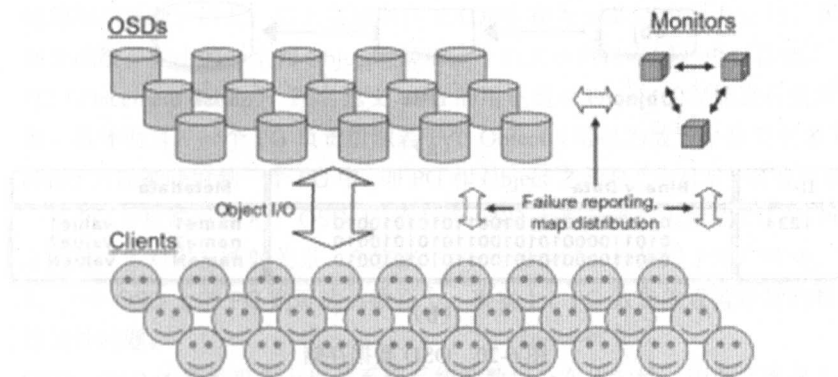


图 6-19 RADOS 结构

在使用 RADOS 系统时, 大量的客户端程序向 Monitor 索取最新的 Cluster Map, 然后直接在本地进行计算, 得出对象的存储位置后, 便直接与对应的 OSD 通信, 完成数据的各种操作。一个 Monitor 集群确保了某个 Monitor 失效时的高可用性。

Ceph 客户端、Monitor 和 OSD 可以相互直接交互, 这意味着 OSD 可以利用本地节点的 CPU 和内存执行那些传统集群架构中有可能拖垮中央服务器的任务, 充分发挥节点上的计算能力。

1. OSD

OSD 用于实现数据的存储于维护。根据定义，OSD 可以被抽象为两个组成部分，即系统部分和守护进程（OSD Deamon）部分。

OSD 的系统部分本质上就是一台安装了操作系统和文件系统的计算机，其硬件部分至少包括一个单核的处理器、一定数量的内存、一块硬盘以及一张网卡。

由于这么小规模 of x86 架构服务器并不实用（事实上也见不到），因而实际应用中通常将多个 OSD 集中部署在一台更大规模的服务器上。在选择系统配置时，应当能够保证每个 OSD 占用一定的计算能力、一定量的内存和一块硬盘（通常情况下一个 OSD 对应一块硬盘）。同时，应当保证该服务器具备足够的网络带宽。

在上述系统平台上，每个 OSD 拥有一个自己的 OSD Deamon。这个 Deamon 负责完成 OSD 的所有逻辑功能，包括与 Monitor 和其他 OSD（事实上是其他 OSD 的 Deamon）通信以维护更新系统状态，与其他 OSD 共同完成数据的存储和维护，与 Client 通信完成各种数据对象操作等。

RADOS 集群从 Ceph 客户端接收数据——不管是来自 Ceph 块设备、Ceph 对象存储、Ceph 文件系统，还是基于 librados 的自定义实现——并存储为对象。如图 6-20 所示，每个对象是文件系统中的文件，它们存储在 OSD 的存储设备上，由 OSD 守护进程处理存储设备上的读/写操作。

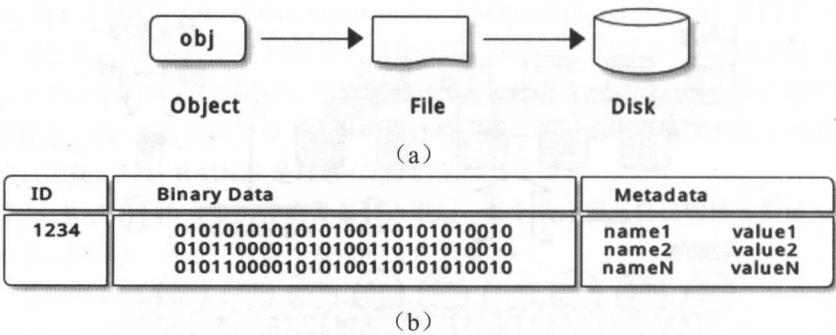


图 6-20 OSD 数据存储

OSD 在扁平的命名空间内把所有数据存储为对象（也就是没有目录层次）。对象包含一个标识符、二进制数据和由“名字/值”对组成的元数据，元数据语义完全取决于 Ceph 客户端。比如，CephFS 用元数据存储文件属性，如文件所有者、创建日期、最后修改日期等。

2. 数据寻址

如前所述，一个大规模分布式存储系统，必须要能够解决两个最基本的问题，即“我应该把数据写入到什么地方”与“我之前把数据写到什么地方去了”，不可避免都涉及数据如何

寻址的问题。Ceph 中的寻址流程如图 6-21 所示。

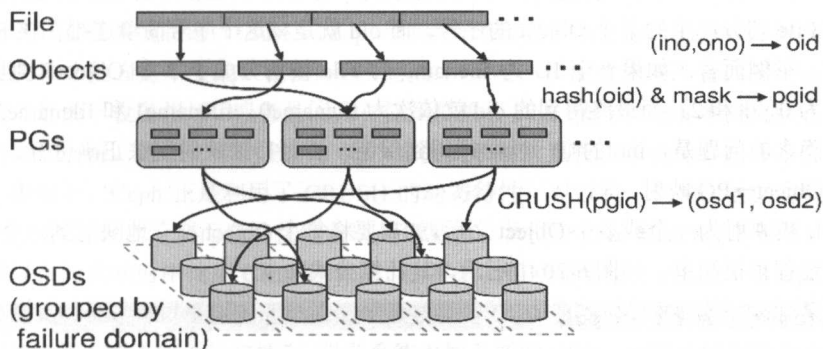


图 6-21 Ceph 寻址流程

- **File:** 此处的 File 就是用户需要存储或者访问的文件。对于一个基于 Ceph 开发的对象存储应用而言, 这个 File 也就对应于应用中的“对象”, 也就是用户直接操作的“对象”。
- **Object:** 此处的 Object 是 RADOS 所看到的“对象”。Object 与上面提到的 File 的区别是, Object 的最大 size 由 RADOS 限定 (通常为 2MB 或 4MB), 以便实现底层存储的组织管理。因此, 当上层应用向 RADOS 存入 size 很大的 File 时, 需要将 File 切分成统一大小的一系列 Object (最后一个的大小可以不同) 进行存储。
- **PG (Placement Group):** 顾名思义, PG 的用途是对 Object 的存储进行组织和位置映射。具体而言, 一个 PG 负责组织若干个 Object (可以为数千个甚至更多), 但一个 object 只能被映射到一个 PG 中, 即 PG 和 Object 之间是“一对多”映射关系。同时, 一个 PG 会被映射到 n 个 OSD 上, 而每个 OSD 上都会承载大量的 PG, 即 PG 和 OSD 之间是“多对多”映射关系。在实践中, n 至少为 2, 如果用于生产环境, 则至少为 3。一个 OSD 上的 PG 则可达到数百个。事实上, PG 数量的设置牵扯到数据分布的均匀性问题。
- **OSD:** OSD 的数量事实上也关系到系统的数据分布均匀性, 因此其数量不应太少。在实践当中, 至少也应该是数十上百个的量级才有助于 Ceph 系统的设计发挥其应有的优势。

(1) File→Object 映射

这次映射的目的是, 将用户要操作的 File, 映射为 RADOS 能够处理的 Object。其映射十分简单, 本质上就是按照 Object 的最大 size 对 file 进行切分, 相当于 RAID 中的条带化过程。这种切分的好处有二: 一是让大小不限的 File 变成最大 size 一致, 可以被 RADOS 高效管理的 Object; 二是让对单一 File 实施的串行处理变为对多个 object 实施的并行化处理。

每一个切分后产生的 Object 将获得唯一的 oid, 即 Object ID。其产生方式也是线性映射, 极其简单。图 6-20 中, ino 是待操作 File 的元数据, 可以简单理解为该 File 的唯一 ID。ono 则是由该 File 切分产生的某个 Object 的序号。而 oid 就是将这个序号简单连缀在该 File ID 之后得到的。举例而言, 如果一个 ID 为 filename 的 File 被切分成了 3 个 Object, 则其 Object 序号依次为 0、1 和 2, 而最终得到的 oid 就依次为 filename0、filename1 和 filename2。

这里隐含的问题是, ino 的唯一性必须得到保证, 否则后续映射无法正确进行。

(2) Object→PG 映射

在 File 被映射为一个或多个 Object 之后, 就需要将每个 Object 独立地映射到一个 PG 中。这个映射过程也很简单, 如图 6-20 中所示, 其计算公式如下:

```
hash(oid) & mask -> pgid
```

由此可见, 其计算由两步组成。首先是使用 Ceph 系统指定的一个静态哈希函数计算 oid 的哈希值, 将 oid 映射成为一个近似均匀分布的伪随机值。然后, 将这个伪随机值和 mask 按位相与, 得到最终的 PG 序号 (pgid)。根据 RADOS 的设计, 给定 PG 的总数为 m (m 应该为 2 的整数幂), 则 mask 的值为 $m-1$ 。因此, 哈希值计算和按位与操作的整体结果事实上是从所有 m 个 PG 中近似均匀地随机选择一个。基于这一机制, 当有大量 Object 和大量 PG 时, RADOS 能够保证 object 和 PG 之间的近似均匀映射。又因为 Object 是由 File 切分而来, 大部分 Object 的 size 相同, 因而, 这一映射最终保证了, 各个 PG 中存储的 Object 的总数据量近似均匀。

这里反复强调了“大量”, 只有当 Object 和 PG 的数量较多时, 这种伪随机关系的近似均匀性才能成立, Ceph 的数据存储均匀性才有保证。为保证“大量”的成立, 一方面, Object 的最大 size 应该被合理配置, 以使得同样数量的 File 能够被切分成更多的 Object; 另一方面, Ceph 也推荐 PG 总数应该为 OSD 总数的数百倍, 以保证有足够数量的 PG 可供映射。

(3) PG→OSD 映射

第三次映射就是将作为 Object 的逻辑组织单元的 PG 映射到数据的实际存储单元 OSD。如图 6-20 所示, RADOS 采用一个名为 CRUSH 的算法, 将 pgid 代入其中, 然后得到一组共 n 个 OSD。这 n 个 OSD 共同负责存储和维护一个 PG 中的所有 Object。前面提到过, n 的数值可以根据实际应用中对于可靠性的需求而配置, 在生产环境下通常为 3。具体到每个 OSD, 则由其上运行的 OSD Daemon 负责执行映射到本地的 Object 在本地文件系统中的存储、访问、元数据维护等操作。

和“Object→PG”映射中采用的哈希算法不同, 这个 CRUSH 算法的结果不是绝对不变的, 而是受到其他因素的影响。其影响因素主要有二:

一是当前系统状态, 也就是在前面有所提及的 Cluster Map (集群运行图)。当系统中的 OSD 状态、数量发生变化时, Cluster Map 可能发生变化, 而这种变化将会影响到 PG 与 OSD 之间的映射。

二是存储策略配置。这里的策略主要与安全相关。利用策略配置, 系统管理员可以指定

承载同一个 PG 的 3 个 OSD 分别位于数据中心的不同服务器乃至机架上，从而进一步改善存储的可靠性。

因此，只有在系统状态（Cluster Map）和存储策略都不发生变化的时候，PG 和 OSD 之间的映射关系才是固定不变的。在实际使用当中，策略一经配置通常不会改变。而系统状态的改变或者是由于设备损坏，或者是因为存储集群规模扩大。好在 Ceph 本身提供了对于这种变化的自动化支持，因而，即便 PG 与 OSD 之间的映射关系发生了变化，也并不会对应用造成困扰。事实上，Ceph 正是利用了 CRUSH 的动态特性，可以将一个 PG 根据需要动态迁移到不同的 OSD 组合上，从而自动化地实现高可靠性、数据分布 re-blancing 等特性。

之所以在此次映射中使用 CRUSH 算法，而不是其他哈希算法，原因之一正是 CRUSH 具有上述可配置特性，可以根据管理员的配置参数决定 OSD 的物理位置映射策略；另一方面是因为 CRUSH 具有特殊的“稳定性”，即当系统中加入新的 OSD，导致系统规模增大时，大部分 PG 与 OSD 之间的映射关系不会发生改变，只有少部分 PG 的映射关系会发生变化并引发数据迁移。这种可配置性和稳定性都不是普通哈希算法所能提供的。因此，CRUSH 算法的设计也是 Ceph 的核心内容之一。

至此为止，Ceph 通过 3 次映射，完成了从 File 到 Object、PG 和 OSD 整个映射过程。通观整个过程，可以看到，这里没有任何的全局性查表操作需求。至于唯一的全局性数据结构 Cluster Map，它的维护和操作都是轻量级的，不会对系统的可扩展性、性能等因素造成不良影响。

接下来的一个问题是：为什么需要引入 PG 并进而而在 Object 与 OSD 之间增加一层映射？

可以想象一下，如果没有 PG 这一层映射，又会怎么样呢？在这种情况下，一定需要采用某种算法，将 Object 直接映射到一组 OSD 上。如果这种算法是某种固定映射的哈希算法，则意味着一个 Object 将被固定映射在一组 OSD 上，当其中一个或多个 OSD 损坏时，Object 无法被自动迁移至其他 OSD 上（因为映射函数不允许），当系统为了扩容新增了 OSD 时，Object 也无法被 re-balance 到新的 OSD 上（同样因为映射函数不允许）。这些限制都违背了 Ceph 系统高可靠性、高自动化的设计初衷。

如果采用一个动态算法（例如仍然采用 CRUSH 算法）来完成这一映射，似乎是可以避免静态映射导致的问题。但是，其结果将是各个 OSD 所处理的本地元数据量暴增，由此带来的计算复杂度和维护工作量也是难以承受的。

例如，在 Ceph 的现有机制中，一个 OSD 平时需要和与其共同承载同一个 PG 的其他 OSD 交换信息，以确定各自是否工作正常，是否需要维护操作。由于一个 OSD 上大约承载数百个 PG，每个 PG 内通常有 3 个 OSD，因此，一段时间内，一个 OSD 大约需要进行数百至数千次 OSD 信息交换。

然而，如果没有 PG 的存在，则一个 OSD 需要和与其共同承载同一个 Object 的其他 OSD 交换信息。由于每个 OSD 上承载的 Object 很可能高达数百万个，因此，同样长度的一段时间内，一个 OSD 大约需要进行的 OSD 间信息交换将暴涨至数百万乃至数千万次。而这种状态

维护成本显然过高。

综上所述，引入 PG 的好处至少有两个：一方面实现了 Object 和 OSD 之间的动态映射，从而为 Ceph 的可靠性、自动化等特性的实现留下了空间；另一方面也有效简化了数据的存储组织，大大降低了系统的维护管理开销。理解这一点，对于彻底理解 Ceph 的对象寻址机制，是十分重要的。

这种分层或分级的设计思路在很多复杂系统的寻址问题上都有应用，比如操作系统里的内存管理多级页表的使用，Intel MPX (Memory Protection Extensions) 技术里 Bound Directory 的引入等。

3. 存储池 (Pool)

存储池是一个逻辑概念，是对存储对象的逻辑分区。Ceph 安装后，会有一个默认的存储池，用户也可以自己创建新的存储池。如图 6-22 所示，一个存储池包含若干 PG 及其所存的若干对象。

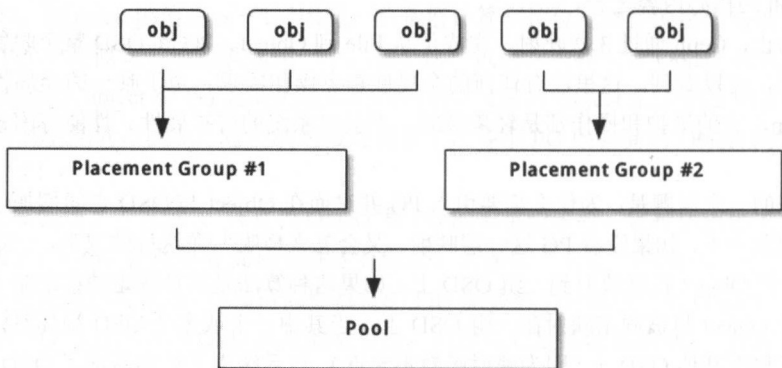


图 6-22 存储池

Ceph 客户端从监视器获取一张集群运行图，并把对象写入存储池。存储池的 size 或副本数、CRUSH 存储规则和归置组数量决定着 Ceph 如何放置数据。我们可以使用以下命令来创建存储池：

```
ceph osd pool create {pool-name} {pg-num} [{pgp-num}] [replicated] \
    [crush-ruleset-name]
ceph osd pool create {pool-name} {pg-num} {pgp-num} erasure \
    [erasure-code-profile] [crush-ruleset-name]
```

从中可以看出，存储池支持：

- 设置数据存储的方法属于多副本还是纠删码。如果是多副本模式，则可以设置副本的数量；如果是纠删码模式，则可以设置数据块和非数据块的数量（纠删码存储池把各对象存储为 $K+M$ 个数据块，其中有 K 个数据块和 M 个编码块）。默认情况下，

为副本模式（即存储每个对象的若干副本），副本数为 3，每个 PG 映射到 3 个 OSD 节点。换句话说，对于每个映射到该 PG 的对象，其数据存储在对应的 3 个 OSD 节点。

- 设置 PG 的数目。合理设置 PG 的数目，可以使资源得到较优的均衡。
- 设置 PGP 的数目。通常情况下，跟 PG 数目一致。当需要增加 PG 数目时，用户数据不会发生迁移，只有进一步增加 PGP 数目时，用户数据才会开始迁移。
- 针对不同的存储池设置不同的 CRUSH 存储规则。比如可以创建规则，指定在选择 OSD 时，选择拥有 SSD 的 OSD 节点。

另外，通过存储池，还可以：

- 提供针对存储池的功能，比如存储池快照等。
- 设置对象的所有者/访问权限。

我们看到这里在 PG 的基础上多出了 PGP 的概念，至于 PG 与 PGP 之间的区别，可以先看“Learning Ceph”和“Ceph Cookbook”作者 Karan Singh 的一段解释。

```
PG = Placement Group
PGP = Placement Group for Placement purpose
pg_num = number of placement groups mapped to an OSD
When pg_num is increased for any pool, every PG of this pool splits into
half, but they all remain mapped to their parent OSD.
Until this time, Ceph does not start rebalancing. Now, when you increase
the pgp_num value for the same pool, PGs start to migrate from the parent to
some other OSD, and cluster rebalancing starts. This is how PGP plays an important
role.
```

总结来说，就是 PG 的增加会引起 PG 内的数据进行分裂，分裂到相同的 OSD 上新生成的 PG 当中，而 PGP 的增加会引起部分 PG 的分布进行变化，但是不会引起 PG 内对象的变动。PGP 相当于存储池 PG 的 OSD 分布的组合个数。

4. CRUSH 算法

在分布式存储系统中面临的一个重要问题是如何在多个存储节点上分布数据。数据分布算法至少要考虑以下 3 个因素：

- 故障域（Failure Domain）隔离。同份数据的不同副本分布在不同的故障域，降低数据损坏的风险。
- 负载均衡。数据能够均匀地分布在磁盘容量不等的存储节点，避免部分节点空闲部分节点超载，从而影响系统性能。
- 控制节点加入离开时引起的数据迁移量。当节点离开时，最优的数据迁移是只有离线节点上的数据被迁移到其他节点，而正常工作的节点的数据不会发生迁移。

一致性 Hash 和 Ceph 的 CRUSH 算法是使用得比较多的数据分布算法。Amazon 的 Dynamo 键值存储系统与 Swift 即使用了一致性 Hash 算法。一致性 Hash 的核心思想是将 Hash 结果域

做成一个空间，通常这个虚拟空间可以描述成一个哈希环，所有存储节点是这个环上的一个点，如图 6-5 所示。需要写入一个对象时，计算对象名称得到的 Hash 值肯定会属于这个 Hash 值空间，也就是说在 Hash 环上面肯定可以找到一个对应的点。比如，这个点位于节点 1 和 2 之间，按照协议，可以选择顺时针离 Hash 值最近的节点作为数据存储点，即新写入的对象可以存入节点 1。

一致性 Hash 算法的最大优点在于可以避免添加存储节点之后的大规模数据迁移。比如，如果后来在节点 1 和节点 2 之间添加了一个节点 8，那么原先存入节点 1 中的一部分数据需要迁移到节点 8，但是，其余节点不需要做任何的数据迁移操作。

但一致性 hash 的一个问题是，存储节点很难将 Hash 空间分布地足够均匀，这样就会导致部分节点负载过重。为了尽可能地解决这一问题，虚拟节点被引入。虚拟节点是相对于物理存储节点而言的，可以相当于物理存储节点的复制品。一个物理节点对应了若干个虚拟节点，虚拟节点的数量是由它自己的容量决定的，虚拟节点负责的分区上的数据最终存储到其对应的物理节点。在一致性 Hash 中引入虚拟节点可以把 Hash 空间划分成更多的分区，从而让数据在存储节点上的分布更加均匀。如图 6-23 所示， Ni_0 代表该虚拟节点对应于物理节点 i 的第 0 个虚拟节点。增加虚拟节点后，物理节点 N0 负责 $[N1_0, N0]$ 和 $[N0, N0_0]$ 两个分区，物理节点 N1 负责 $[N0_0, N1]$ 和 $[N2_0, N1_0]$ 两个分区，物理节点 N2 负责 $[N2, N1]$ 和 $[N2_0, N2]$ 两个分区，3 个物理节点负责的总的数据量趋于平衡。

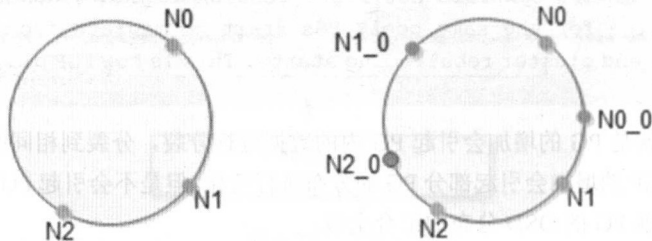


图 6-23 虚拟节点

虚拟节点的引入能够使每个节点负责多个部分的数据，这样一个节点移除后，它所负责的多个分区会托管给多个节点处理，因此这种思想一定程度上解决了数据分布不均的问题。

实际应用中，可以根据物理节点的磁盘容量的大小来确定其对应的虚拟节点数目。虚拟节点数目越多，节点负责的数据区间也越大。

当节点加入或者离开时，分区会相应地进行分裂或合并。这对新写入的数据不构成影响，但对已经写入到磁盘的数据需要重新计算 Hash 值，以确定它是否需要迁移到其他节点。因为需要遍历磁盘中的所有数据，这个计算过程非常耗时。为了解决这一问题，Dynamo 系统将分区和分区位置（也就是分区所对应的物理存储节点）进行分离，将 Hash 空间划分成固定的若干个分区，并维持分区数目和虚拟节点数目相等，每个虚拟节点负责一个分区，比如图 6-24a

所示的固定分区[A, B], [B, C], [C, D]和[D, A], 以及对应的虚拟节点 (Token) T0, T1, T2 和 T3。由于分区固定, 因此迁移数据时可以比较容易知道哪些数据需要迁移, 哪些数据不需要迁移。

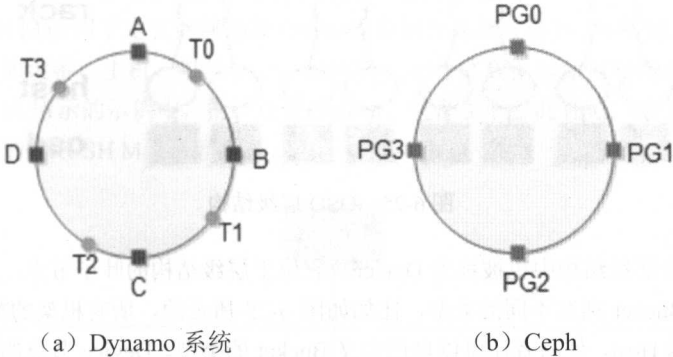


图 6-24 固定分区

而对于 CRUSH 算法来说, PG 的引入也类似于 Dynamo 这种划分固定分区的思想, 如图 6-23b 所示。PG 是抽象的存储节点, 它不会随着物理节点的加入或离开而增加或减少, 对象到 PG 的映射是稳定的。所有的 PG 相当于把 Hash 环划分成固定的分区, 每个 PG 管理的数据区间相同, 因而数据能够均匀地分布到 PG 上。同时, PG 又充当了 Dynamo 系统中虚拟节点的角色, 决定了分区对应的物理存储位置 (PG→OSD 的映射)。

CRUSH 算法的目的是, 为给定的 PG (即分区) 分配一组存储数据的 OSD 节点。如图 6-20 中 PG→OSD 映射的计算公式:

$$\text{CRUSH}(\text{pgid}) \rightarrow (\text{osd0}, \text{osd1}, \text{osd2}, \dots, \text{osdn})$$

选择 OSD 节点的过程, 要考虑以下几个因素:

- PG 在 OSD 间均匀分布。假设每个 OSD 的磁盘容量都相同, 那么我们希望 PG 在每个 OSD 节点上是均匀分布的, 也就是说每个 OSD 节点包含相同数目的 PG。假如节点的磁盘容量不等, 那么容量大的磁盘的节点能够处理更多数量的 PG。
- PG 的 OSD 分布在不同的故障域。因为 PG 的 OSD 列表用于保存数据的不同副本, 副本分布在不同的 OSD 中可以降低数据损坏的风险。

Ceph 使用树形层级结构描述 OSD 的空间位置以及权重 (同磁盘容量相关) 大小。如图 6-25 所示, 层级结构描述了 OSD 所在主机、主机所在机架以及机架所在机房等空间位置。这些空间位置隐含了故障区域, 例如使用不同电源的不同的机架属于不同的故障域。CRUSH 能够依据一定的规则将副本放置在不同的故障域。

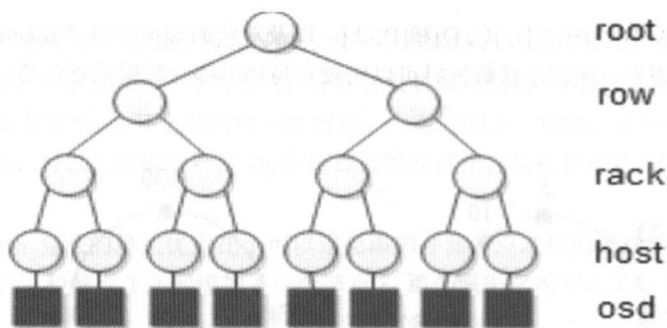


图 6-25 OSD 层级结构

OSD 节点在层级结构中也被称为 Device，它位于层级结构的叶子节点，所有非叶子节点称为 Bucket。Bucket 拥有不同的类型，比如如图 6-25 所示的，所有机架的类型为 Rack，所有主机的类型为 Host。使用者还可以自己定义 Bucket 的类型。Device 节点的权重代表存储节点的性能，磁盘容量是影响权重大小的重要参数。Bucket 节点的权重是其子节点的权重之和。

CRUSH 算法通过每个设备的权重来计算数据对象的分布。对象分布是由 Cluster Map 和 Data Distribution Policy 决定的。Cluster Map 描述了可用存储资源和层级结构（比如有多少个机架，每个机架上有多少个服务器，每个服务器上有多少个磁盘）。Data Distribution Policy 由 Placement Rules 组成，决定了每个数据对象有多少个副本，这些副本存储的限制条件（比如 3 个副本放在不同的机架中）。

CRUSH 只使用了 Cluster Map、Placement Rules 和 pgid，利用多参数 Hash 函数，Hash 函数中的参数包括 pgid，使得从 PG 到 OSD 集合是确定性的和独立的。CRUSH 是伪随机算法，相似输入的结果之间没有相关性。

5. Monitor

Ceph 客户端读或写数据前必须先连接到某个 Ceph 监视器，获得最新的集群运行图副本。一个 Ceph 存储集群只需要单个监视器就能运行，但它就成了单一故障点（即如果此监视器宕机，Ceph 客户端就不能读写数据了）。为增强可靠性和容错能力，Ceph 支持监视器集群；在一个监视器集群内，延时以及其他错误会导致一到多个监视器滞后于集群的当前状态，因此，Ceph 的各监视器例程必须就集群的当前状态达成一致。

由若干个 Monitor 组成的监视器集群共同负责整个 Ceph 集群中所有 OSD 状态的发现与记录，并且形成 Cluster Map 的主拷贝，包括集群成员、状态、变更，以及 Ceph 存储集群的整体健康状况。随后，这份 Cluster Map 被扩散至全体 OSD 以及 Client。OSD 使用 Cluster Map 进行数据的维护，而 Client 使用 Cluster Map 进行数据的寻址。

在集群中，各个 Monitor 的功能总体上是一样的，其相互间的关系可以被简单理解为主从备份关系。Monitor 并不主动轮询各个 OSD 的当前状态。正相反，OSD 需要向 Monitor 上报

状态信息。常见的上报有两种情况：一是新的 OSD 被加入集群，二是某个 OSD 发现自身或者其他 OSD 发生异常。在收到这些上报信息后，Monitor 将更新 Cluster Map 信息并加以扩散。

Cluster Map 实际上是多个 Map 的统称，包括 Monitor Map、OSD Map、PG Map、CRUSH Map 以及 MDS Map 等，各运行图维护着各自运营状态的变更。

其中 CRUSH Map 用于定义如何选择 OSD，内容包含存储设备列表、故障域树状结构（设备的分组信息，如设备、主机、机架、行、房间等）和存储数据时如何利用此树状结构的规则。比如如图 6-26 所示的示例中，根节点是 Default，包含 3 个主机 Host，每个 Host 包含 3 个 OSD 服务，相应的 CRUSH Map 片段如下：

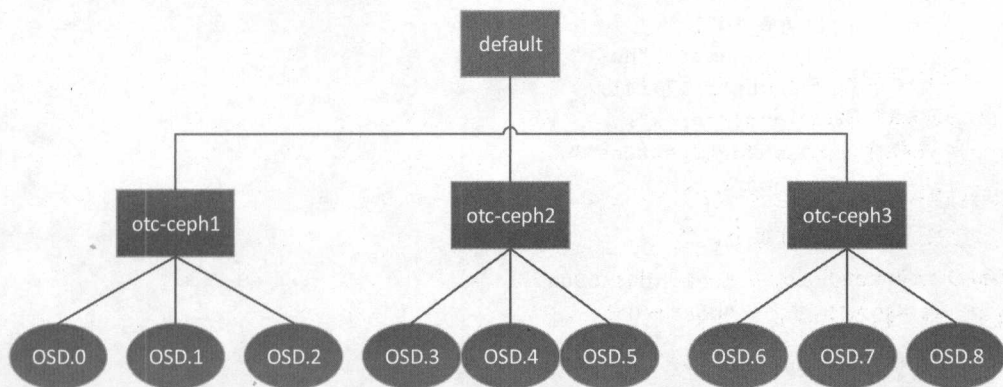


图 6-26 CRUSH map 示例

```
"buckets": [
  {
    "id": -1,
    "name": "default",
    "type_id": 10,
    "type_name": "root",
    "weight": 280859,
    "alg": "straw",
    "hash": "rjenkins1",
    "items": [
      {
        "id": -2,
        "weight": 177209,
        "pos": 0
      },
      {
        "id": -3,
        "weight": 86376,
        "pos": 1
      }
    ]
  }
]
```

```

    },
    {
      "id": -4,
      "weight": 17274,
      "pos": 2
    }
  ],
  {
    "id": -2,
    "name": "otc-ceph3",
    "type_id": 1,
    "type_name": "host",
    "weight": 177141,
    "alg": "straw",
    "hash": "rjenkins1",
    "items": [
      {
        "id": 0,
        "weight": 59047,
        "pos": 0
      },
      {
        "id": 1,
        "weight": 59047,
        "pos": 1
      },
      {
        "id": 2,
        "weight": 59047,
        "pos": 2
      }
    ]
  },
  {
    "id": -3,
    "name": "otc-ceph2",
    "type_id": 1,
    "type_name": "host",
    "weight": 86310,
    "alg": "straw",
    "hash": "rjenkins1",
    "items": [
      {

```

```

        "id": 3,
        "weight": 28770,
        "pos": 0
    },
    {
        "id": 4,
        "weight": 28770,
        "pos": 1
    },
    {
        "id": 5,
        "weight": 28770,
        "pos": 2
    }
]
},
{
    "id": -4,
    "name": "otc-ceph4",
    "type_id": 1,
    "type_name": "host",
    "weight": 17274,
    "alg": "straw2",
    "hash": "rjenkins1",
    "items": [
        {
            "id": 6,
            "weight": 5957,
            "pos": 0
        },
        {
            "id": 7,
            "weight": 5957,
            "pos": 1
        },
        {
            "id": 8,
            "weight": 5360,
            "pos": 2
        }
    ]
}
}
1,

```

如前所述，在这个树形结构中，所有非叶子节点称为 Bucket，所有 Bucket 的 ID 号都是

负数，便于和 OSD 的 ID 进行区分。选择 OSD 时，需要先指定一个 Bucket，然后选择它的一个子 Bucket，这样一级一级递归，直到到达设备（叶子）节点。目前有 5 种算法来实现子节点的选择：Uniform、List、Tree、Straw 和 Straw2。如表 6-1 所示，这些算法的选择影响了两个方面的复杂度：在一个 Bucket 中，找到对应的节点的复杂度以及当一个 Bucket 中的 OSD 节点丢失或者增加时，数据移动的复杂度。

表 6-1 不同 Bucket 算法复杂度的比较

操作	Uniform	List	Tree	Straw/Straw2
查找	O(1)	O(n)	O(lgn)	O(n)
增加	Poor	Optimal	Good	Optimal
删除	Poor	Poor	Good	Optimal

其中，Uniform 适用于 item 具有相同权重，而且 Bucket 很少有添加或删除 item 的情况，它的查找速度是最快的。Straw/Straw2 不像 List 和 Tree 一样都需要遍历，而是让 Bucket 所包含的所有 item 公平的竞争，这种算法就像抽签一样，所有的 item 都有机会被抽中（只有最长的签才能被抽中，每个签的长度与权重有关）。

除了存储设备的列表以及树状结构之外，CRUSH Map 还包含了存储规则用来指定在每个存储池中选择特定 OSD 的 Bucket 范围，同时可以指定备份的分布规则。默认情况下，CRUSH Map 有一个存储规则，如果用户创建存储池时没有指定 CRUSH 规则就使用该规则。但是用户可以自己定义 CRUSH 规则，指定给特定存储池。

下面是默认的 CRUSH 规则，重点在 steps 部分。这里指定从 default 这个 Bucket 开始，选择 3 个（Pool 创建时指定的副本数）Host，在这 3 个 Host 中再选择 OSD。每个对象的 3 份数据将位于 3 个不同的主机上。

```
"rules": [
  {
    "rule_id": 0,
    "rule_name": "replicated_ruleset",
    "ruleset": 0,
    "type": 1,
    "min_size": 1,
    "max_size": 10,
    "steps": [
      {
        "op": "take",
        "item": -1,
        "item_name": "default"
      },
      {
        "op": "chooseleaf_firstn",
        "num": 0,
```

```
        "type": "host"
    },
    {
        "op": "emit"
    }
  ]
},
```

6. 数据操作流程

Ceph 的读写操作采用 Primary-Replica 模型，Client 只向 Object 所对应 OSD set 的 Primary 发起读写请求，这保证了数据的强一致性。当 Primary 收到 Object 的写请求时，它负责把数据发送给其他 Replicas，只有这个数据被保存在所有的 OSD 上时，Primary 才应答 Object 的写请求，这保证了副本的一致性。

这里以 Object 写入为例，假定一个 PG 被映射到 3 个 OSD 上。对象写入流程如图 6-27 所示。

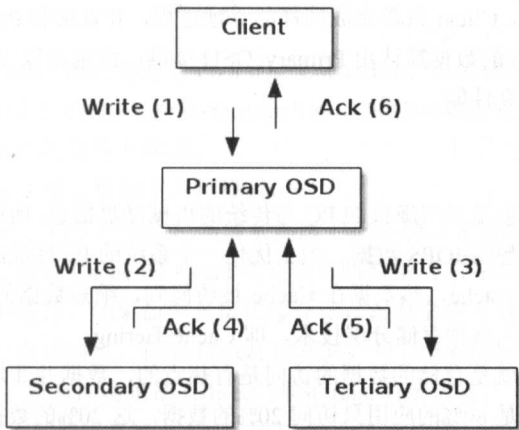


图 6-27 对象写入流程

当某个 Client 需要向 Ceph 集群写入一个 File 时，首先需要在本地完成前面所述的寻址流程，将 File 变为一个 Object，然后找出存储该 Object 的一组 3 个 OSD。这 3 个 OSD 具有各自不同的序号，序号最靠前的那个 OSD 就是这一组中的 Primary OSD，而后两个则依次是 Secondary OSD 和 Tertiary OSD。

找出 3 个 OSD 后，Client 将直接和 Primary OSD 通信，发起写入操作（步骤 1）。Primary OSD 收到请求后，分别向 Secondary OSD 和 Tertiary OSD 发起写入操作（步骤 2 和步骤 3）。当 Secondary OSD 和 Tertiary OSD 各自完成写入操作后，将分别向 Primary OSD 发送确认信息（步骤 4 和步骤 5）。当 Primary OSD 确信其他两个 OSD 的写入完成后，则自己也完成数据

写入，并向 Client 确认 Object 写入操作完成（步骤 6）。

之所以采用这样的写入流程，本质上是为了保证写入过程中的可靠性，尽可能避免造成数据丢失。同时，由于 Client 只需要向 Primary OSD 发送数据，因此，在 Internet 使用场景下的外网带宽和整体访问延迟又得到了一定程度的优化。

当然，这种可靠性机制必然导致较长的延迟，特别是，如果等到所有的 OSD 都将数据写入磁盘后再向 Client 发送确认信号，则整体延迟可能难以忍受。因此，Ceph 可以分两次向 Client 进行确认。当各个 OSD 都将数据写入内存缓冲区后，就先向 Client 发送一次确认，此时 Client 即可以向下执行。待各个 OSD 都将数据写入磁盘后，会向 Client 发送一个最终确认信号，此时 Client 可以根据需要删除本地数据。

分析上述流程可以看出，在正常情况下，Client 可以独立完成 OSD 寻址操作，而不必依赖于其他系统模块。因此，大量的 Client 可以同时和大量的 OSD 进行并行操作。同时，如果一个 File 被切分成多个 Object，这多个 Object 也可被并行发送至多个 OSD。

从 OSD 的角度来看，由于同一个 OSD 在不同的 PG 中的角色不同，因此，其工作压力也可以被尽可能均匀地分担，从而避免单个 OSD 变成性能瓶颈。

如果需要读取数据，Client 只需完成同样的寻址过程，并直接和 Primary OSD 联系。目前的 Ceph 设计中，被读取的数据默认由 Primary OSD 提供，但也可以设置允许从其他 OSD 获取，以分散读取压力提高性能。

7. Cache Tier

分布式的集群一般都是采用廉价的 PC 与传统的机械硬盘搭建，所以在磁盘的访问速度上有一定的限制，没有理想的 IOPS 数据。当去优化一个系统的 IO 性能时，最先想到的就是添加快速的存储设备作为 Cache，热数据在 Cache 被访问到，缩短数据的访问延时。Ceph 也从 Firefly 0.80 版本开始引入这种存储分层技术，即 Cache Tiering。

分层存储的原理，就是存储的数据的访问是有热点的，数据并非均匀访问。有个通用法则叫做二八原则，也就是 80% 的应用只访问 20% 的数据，这 20% 的数据成为热点数据。如果把这些热点数据保存性能比较高的 SSD 磁盘上，就可以提高响应时间。

Cache Tiering 的基本思想就是冷热数据分离，用相对快速/昂贵的存储设备比如 SSD，组成一个 Pool 来作为 Cache 层，后端用相对慢速/廉价的设备来组建冷数据存储池，作为 Storage 层或者说 Base 层。Cache 层维护有 Base 层的一部分数据，Cache 层需要是多副本模式，Storage 则可以是多副本或者纠删码模式。

在 Cache Tiering 中有一个分层代理，当存在 Cache 层的数据变冷或不再活跃时，该代理把这些数据刷到 Storage 层，最后把它们从 Cache 层中移除，这些操作称为刷新（Flush）和逐出（Evict）。

如图 6-28 所示，Ceph 的对象处理器（Objecter，位于 osdc 即 OSD 客户端模块）决定往哪里存储对象，分层代理决定何时把缓存内的对象刷回后端存储层，所以缓存层和后端存储

层对 Ceph 客户端来说是完全透明的。

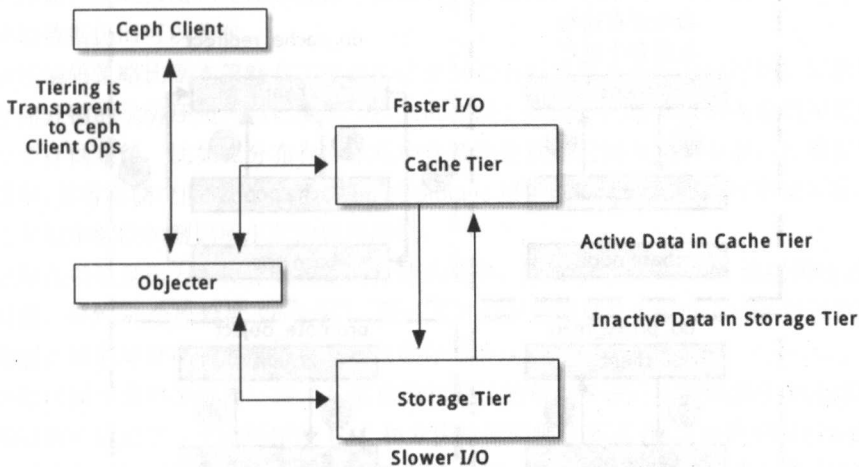


图 6-28 Cache Tier

需要注意的是，Cache Tier 是基于 Pool 的，Cache Pool 对应 Storage Pool，不是 SSD 磁盘对应机械硬盘的，所以在 Cache Tier 和 Storage Tier 之间移动数据是两个 Pool 之间数据的移动，数据可能在不同地点的设备上移动。

目前 Cache Tiering 主要支持如下几种模式。

- writeback 模式：对于写操作，当请求到达 Cache 层，完成写操作后，直接返回给客户端应答，后面由 Cache 的 Agent 线程负责将数据写入 Storage Tier。对于读操作，则看是否命中缓存，如果命中直接在缓存读，没有命中可以 redirect 到 Storage Tier 访问。如果近期访问过，说明 Object 比较热，可以 promote 到 cache 中。
- forward 模式：所有的请求都 redirect 到 Storage Tier 访问。
- readonly 模式：写请求直接 redirect 到 Storage Tier 访问，读请求命中则直接处理，没有命中需要从 Storage Tier 提升（promote）到 Cache Tier 中，完成请求，下次再读取直接命中缓存。
- readforward 模式：读请求都 redirect 到 Storage Tier 中，写请求采用 writeback 模式。
- readproxy 模式：读请求发送给 Cache Tier，Cache Tier 去 Base Pool 中读取，获得 Object 后，Cache Tier 自己不保存，直接发送给客户端，写请求采用 writeback 模式。
- proxy 模式：对于读写请求都是采用 proxy 的方式，不是转发而是代表 Client 去进行操作，Cache Tier 自己并不保存。

这里频繁提及了 redirect、promote 与 proxy 等几种操作，如图 6-29 所示。

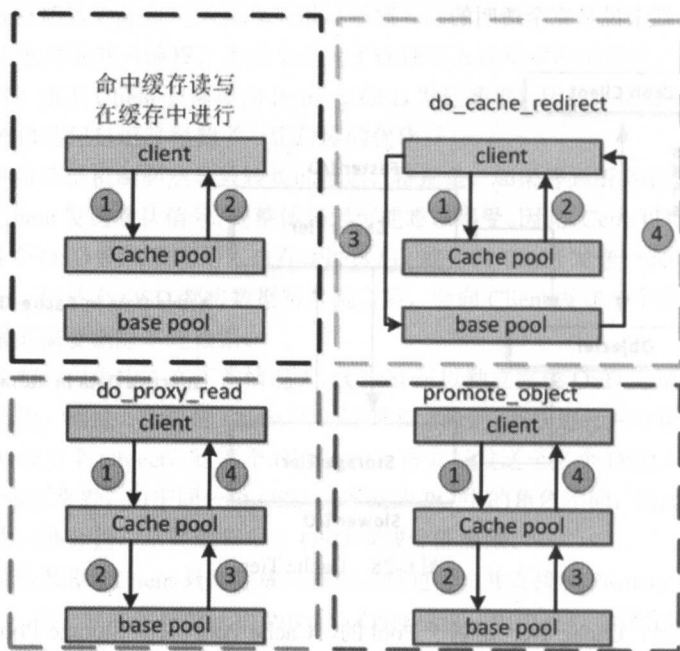


图 6-29 redirect、proxy 与 promote 操作

- **redirect:** 客户端请求 Cache Pool，Cache Pool 告诉客户端应该去 Base Pool 中请求，客户端收到应答后，再次发送请求到 Base Pool 中请求数据，由 Base Pool 告诉客户端请求完成。
- **proxy:** 客户端发送读请求到 Cache Pool，但是未命中，则 Cache Pool 自己会发送请求到 Base Pool 中，获取数据后，由 Cache Pool 将数据发送给客户端，完成读请求。但是值得注意的是，虽然 Cache Pool 读取到了该 Object，但不会保存在 Cache Pool 中，下次请求仍然需要重新向 Base Pool 请求。
- **promote:** 当客户端发送请求到 Cache Pool 中，但是 Cache Pool 未命中，Cache Pool 会选择将该 Object 从 Base Pool 中提升到 Cache Pool 中，然后在 Cache Pool 进行读写操作，操作完成后告知客户端请求完成，在 Cache Pool 会缓存该 Object，下次直接在 Cache 中处理，和 proxy 操作存在区别。

8. 多副本与纠删码

如前所述，Cache Tier 架构里，Cache 层需要是多副本模式，Storage 则可以是多副本或者纠删码模式。而在我们创建一个存储池的时候，也需要设置数据存储的方法属于多副本还是纠删码。这里我们就多副本与纠删码进行更为细致的阐述。

副本策略和编码策略是保证数据冗余度的两个重要方法。当原始数据发生部分丢失时，

副本策略和编码策略都可以保证数据仍旧可以正确获取。副本策略将原始数据复制一份或多份进行存储，编码策略则将原始数据分块并编码生成冗余数据块，保证丢失一定量内的数据块，原始数据仍旧可以获取。

虽然编码策略比副本策略存在更高的计算开销而且修复需要一定的时间，但能够极大地减少存储开销的优势还是为自己赢得了巨大的空间。实际上，副本策略和编码策略也往往共存于一个存储系统，比如在分布存储系统中热数据往往通过副本策略保存，冷数据则通过编码后保存，节省存储空间。比如上述 Ceph 的 Storage 层可以使用纠删码提高存储容量，而 Cache 层使用多副本解决纠删码所引起的速度降低。

比如在以磁盘作为单位存储设备的存储系统中，假设磁盘总数为 n ，编码策略通过编码 k 个数据盘，得到 m 个校验盘 ($n=k+m$)，保证丢失若干个磁盘 (不超过 m 个) 可以恢复出丢失磁盘数据。磁盘可以推广为数据块或者任意存储节点。

纠删码属于编码策略的一种，从信息论和编码的角度来说，纠删码属于分组线性编码，其编码过程可以通过一个编码矩阵 GM 和分块数据的乘法来表示，也就是说编码矩阵 GM 定义了数据是如何编码为冗余数据的。以图 6-30 为例， $C0 \sim C5$ 是冗余数据，所有的冗余数据可以表示为 $GM \times D\{D0、D1、D2、D3\}$ 的乘法，编码矩阵 GM 的列数对应着原始数据分块个数 (k)，行数对应着编码后所有数据块个数 (n)。

$$\begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \\ m_{40} & m_{41} & m_{42} & m_{43} \\ m_{50} & m_{51} & m_{52} & m_{53} \end{bmatrix} \cdot \begin{bmatrix} D0 \\ D1 \\ D2 \\ D3 \end{bmatrix} = \begin{bmatrix} C0 \\ C1 \\ C2 \\ C3 \\ C4 \\ C5 \end{bmatrix}$$

图 6-30 纠删码

以一个包含 5 个 OSD 的纠删码存储池为例，能够容忍 2 个丢失 ($k=3, m=2$)，当包含 ABCDEFGHI 的对象 NYAN 被写入存储池时，纠删编码函数把内容分割为 3 个数据块：第一份包含 ABC，第二份是 DEF，最后是 GHI，若内容长度不是 k 的倍数则需填充。此函数还会创建两个编码块：第四个是 YXY，第五个是 GQC。

对象 NYAN 中的块有相同的名字但存储在不同的 OSD 中，分块的顺序也被保存在对象的属性中。如图 6-31 所示，包含 ABC 的块 1 存储在 OSD5 上，包含 YXY 的块 4 存储在 OSD3 上。

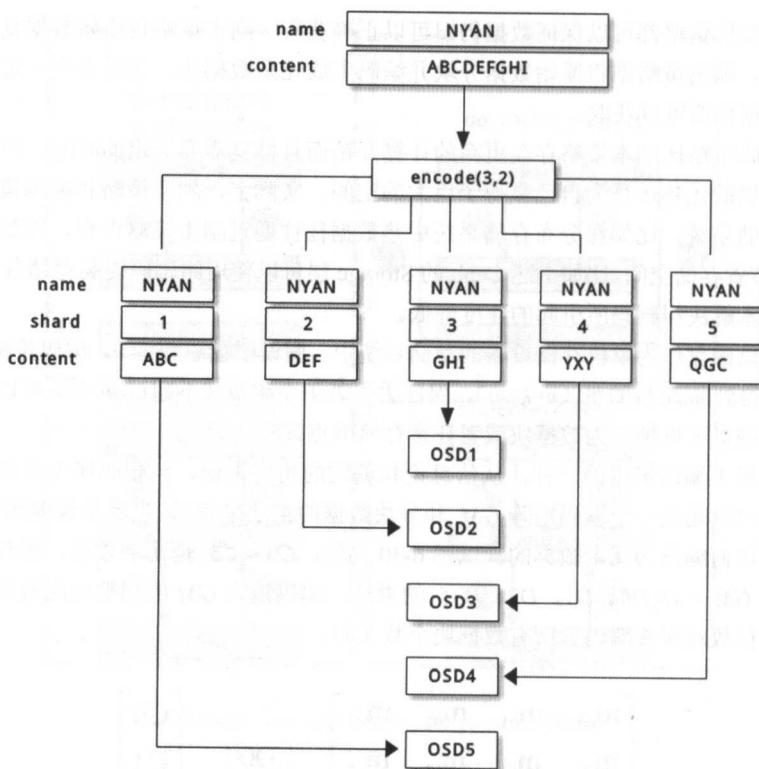


图 6-31 对象 NYAN 被写入纠删码存储池

从纠删码存储池中读取 NYAN 对象时，只要有 3 块读出就可以成功调用解码函数。如图 6-32 所示，解码函数会读取 3 个块：包含 ABC 的块 1，包含 GHI 的块 3 和包含 YXY 的块 4，然后重建对象的原始内容 ABCDEFGHI。解码函数被告知块 2 和 5 丢失了，块 5 不可读是因为 OSD4 损坏，块 2 是因为 OSD2 最慢，其数据未被采纳。

在纠删码存储池中，主 OSD 接受所有的写操作，并负责把数据编码为 $k+m$ 个块并发送给其他 OSD。

纠删码通过技术含量较高的算法，提供和多副本近似的可靠性，同时减小了额外所需冗余设备的数量，从而提高了存储设备的利用率。但纠删码所带来的额外负担主要是计算量和数倍的网络负载，优缺点都相当明显。尤其是在出现硬盘故障后，重建数据非常耗 CPU，而且计算一个数据块需要通过网络读出多倍的数据并传输，所以网络负载也有数倍甚至 10 数倍的增加。

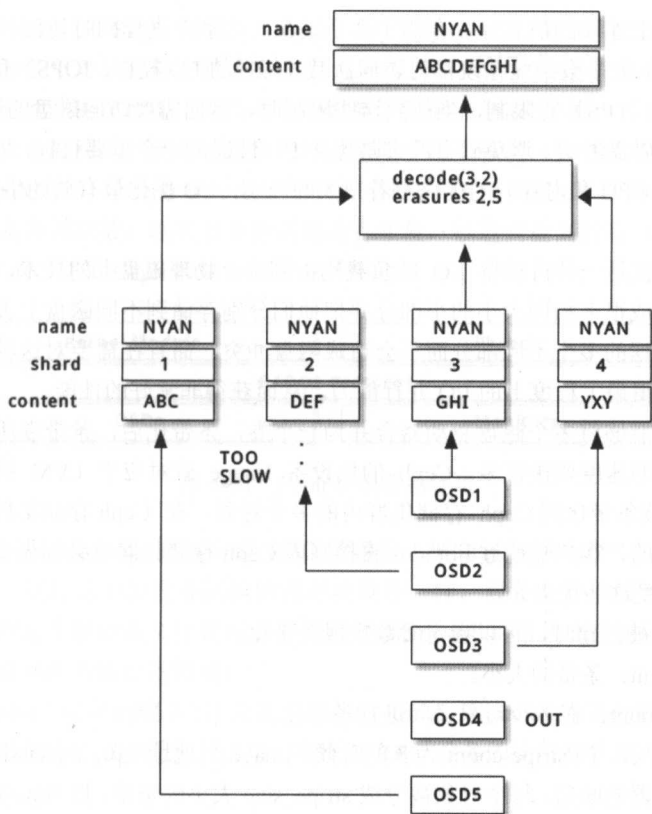


图 6-32 从纠删码存储池读取对象 NYAN

因此，纠删码适合冷数据访问比较少的数据，比如镜像的存储、适合慢的硬件设备、热数据的存储可以通过 Cache 层使用快速设备去存储。

整体来看，若采用纠删码技术，你能够得到所希望的容错能力和存储资源利用率，但是需要接受一定的数据重建代价，两者间做一个平衡。

6.4.3 Ceph 块设备

如前所述，Ceph 可以以一套存储系统同时提供对象存储、块存储和文件系统存储 3 种功能。Ceph 存储集群 RADOS 自身是一个对象存储系统，基础库 librados 提供一系列的 API 接口允许用户操作对象，和 OSD、MON 等进行通信。基于 RADOS 与 librados，Ceph 通过 RBD (Reliable Block Device) 提供了一个标准的块设备接口，提供基于块设备的访问模式。

Ceph 中的块设备称为 Image，是 thin-provisioned 的，即按需分配，大小可调且将数据条带化存储到集群内的多个 OSD。

条带化是指把连续的信息分片存储于多个设备。当多个进程同时访问一个磁盘时，可能会出现磁盘冲突。大多数磁盘系统都对访问次数（每秒的 I/O 操作，IOPS）和数据传输率（每秒传输的数据量，TPS）有限制，当达到这些限制时，后面需要访问磁盘的进程就需要等待，这时就是所谓的磁盘冲突。避免磁盘冲突是优化 I/O 性能的一个重要目标，而 I/O 性能的优化与其他资源（如 CPU 和内存）的优化有着很大的区别，I/O 优化最有效的手段是将 I/O 最大限度地进行平衡。

条带化技术就是一种自动将 I/O 的负载均衡到多个物理磁盘上的技术，条带化技术将一块连续的数据分成很多相同大小的小部分并把他们分别存储到不同磁盘上去。这就能使多个进程同时访问数据的多个不同部分而不会造成磁盘冲突，而且在需要对这种数据进行顺序访问的时候可以获得最大程度上的 I/O 并行能力，从而获得非常好的性能。

条带化技术能够将多个磁盘驱动器合并为一个卷，条带化后，条带卷所能提供的速度比单个盘所能提供的速度要快很多。Ceph 的块设备 Image 就对应于 LVM 的逻辑卷（Logical Volume），可以被条带化到 Ceph 存储集群内的多个对象。在 Ceph 存储集群内存储的那些对象是没被条带化的，客户端通过 librados 直接写入 Ceph 存储集群前必须先自己条带化（和并行 I/O）才能享受这些优势。

Ceph Image 被创建时，可以指定参数实现条带化。

- stripe-unit: 条带的大小。
- stripe-count: 在多少对象之间进行条带化。

如图 6-33 所示，当 stripe-count 为 3 的时候，Image 上地址从[0, object-size*stripe_count-1]的地址到对象位置的映射。每个对象被分成 stripe_size 大小的条带，按 stripe_count 分成一组，Image 在上面依次分布。Image 上[0, stripe_size-1]对应对象 Object1 上的[0, stripe_size-1]，Image 上[stripe_size, 2*stripe_size-1]对应 Object2 上的[0, stripe_size-1]，以此类推。

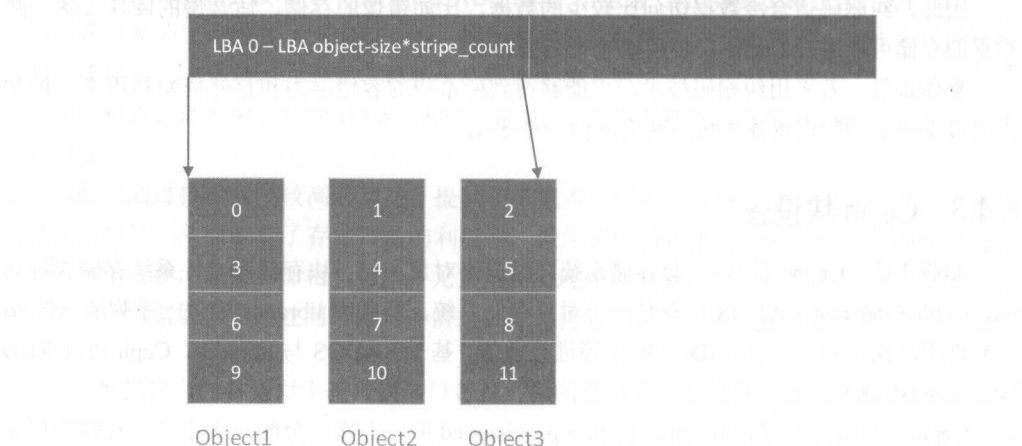


图 6-33 Ceph Image 条带化

处理大尺寸图像、大 Swift 对象（如视频）等的时候，我们能看到条带化到一个对象集（Object Set）中的多个对象能带来显著的读/写性能提升。当客户端把条带单元并行地写入相应对象时，就会有明显的写性能，因为对象映射到了不同的 PG，并进一步映射到不同 OSD，可以并行地以最大速度写入。到单一磁盘的写入受限于磁头移动（如 6ms 寻道时间）和存储设备带宽（如 100MB/s），Ceph 把写入分布到多个对象（它们映射到了不同 PG 和 OSD），这样可减少每设备寻道次数，联合多个驱动器的吞吐量，以达到更高的写（或读）速度。

如图 6-34 所示，使用 Ceph 的块存储有两种路径。

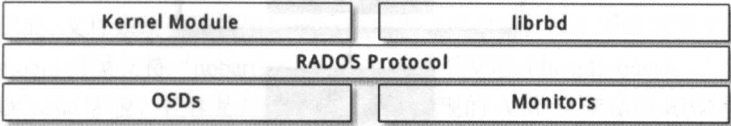


图 6-34 Ceph 块设备的使用

- 通过 Kernel Module: 就是创建 RBD 设备后，把它 map 到内核中，成为一个虚拟的块设备，这时这个块设备同其他通用块设备一样，设备文件一般为/dev/rbd0，后续直接使用这个块设备文件就可以了，既可以把/dev/rbd0 格式化后 mount 到某个目录，也可以直接作为裸设备使用。
- 通过 librbd: 就是创建 RBD 设备后，使用 librbd、librados 库访问管理块设备。这种方式直接调用 librbd 提供的接口，实现对 RBD 设备的访问和管理，不会在客户端产生块设备文件。

第二种方式主要为虚拟机提供块存储设备。在虚拟机场景中，一般会用 Qemu/KVM 中的 RBD 驱动部署 Ceph 块设备，宿主机通过 librbd 向客户机提供块设备服务。QEMU 可以直接通过 librbd 像访问虚拟块设备（Virtual Block Device）一样访问 Ceph Image。。图 6-35 所示为使用 librbd 方式时的 I/O 协议栈。

librbd 使用 RBDCache 在客户端侧缓存数据（相应地对于基于 Kernel Module 的第一种方式，使用 Page Cache 达到同样的目的）。RBDCache 主要提供了读缓存和写合并的功能，用来提高读写性能。

- 写缓存: 启用 RBDCache 时，librbd 将数据写入 RBDCache，然后被 Flush 到 Ceph 集群，其效果就是多个写操作被合并，但是有一定的时间延迟。
- 读缓存: 数据会在缓存中被保留一段时间，这期间 librbd 读数据的话，会直接从缓存中读取，提高读效率。
- 合并写操作: 对同一个 OSD 上的多个写操作，应该会合并为一个大的写操作，提高写入效率。

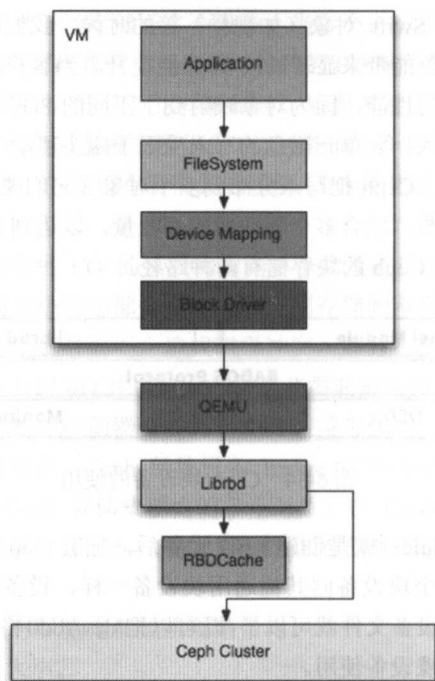


图 6-35 librbd I/O 协议栈

理论上，RBDCache 对顺序写的效率提升应该非常有帮助，而对随机写的效率提升应该没那么大，其原因是后者合并写操作的效率没有前者高（也就是能够合并的写操作的百分比比较少）。

RBDCache 与前面所述的 RADOS 中的 Cache Tier 主要差异在于缓存的位置不同，Cache Tier 是 RADOS 层在 OSD 端进行数据缓存，也就是说不论是块存储、对象存储还是文件存储都可以使用它来提高读写速度，RBDCache 是 RBD 层在客户端的缓存，只支持块存储。

因为 RBDCache 是位于客户端侧的缓存，当多个客户端使用同个块设备时，存在客户端数据不一致的问题。比如，用户 A 向块设备写入数据后，数据停留在客户自己的缓存中，没有立即刷新到磁盘，所以其他用户读取不到 A 写入的数据。但是 Cache Tier 不存在这个问题，因为所有用户的数据都直接写入到 SSD，用户读取数据也是在 SSD 中读取的，所以不存在客户端数据不一致的问题。

通常 Cache Tier 使用 SSD 作缓存，而 RBDCache 使用内存作缓存。SSD 和内存有两个方面的差别，一个是读写速度，另一个是掉电保护。掉电后内存中的数据就丢失了，而 SSD 中的数据不会丢失。所以 RBDCache 需要提供一些策略来不断回写到 Ceph 集群实现持久化，在 librbd 中有若干选项来控制 RBDCache 的大小和回写策略，当满足缓存回写的要求时（空间或者数据保存时间达到阈值）即会回写数据。此外，librbd 也提供了 Flush 接口将缓存中

的脏数据全部回写。

如果将 Ceph 系统与单机系统作类比, RBDCache 则类似于传统磁盘上的控制器缓存(这类缓存不归 Kernel 管理),它们存在的意义一致,也同样面临在机器掉电情况下,缓存数据丢失的情况。但是现代磁盘控制器都会配置一个小型电容用来实现在机器掉电后对缓存数据的回写,但是 Linux Kernel 无法知晓到底是否存在这类“急救”装置来实现持久性,因此,大多数文件系统在实现 `fsync` 这类接口强制执行对某个文件数据的回写时,同时会使用 Kernel Block 模块提供的 API 去给块设备发送一个 Flush Request,块设备收到 Flush Request 后就会回写自身的缓存。但是如果机器上存在电容,那么实际上 Flush Request 会大大降低文件系统的读写性能,因此,文件系统会提供如 `barrier` 选项来让用户选择是否需要发送 Flush Request,比如 XFS 在 mount 时就支持“`nobarrier`”选项来选择 not 发送 Flush Request。

对于 RBDCache 来说,往往是在使用 QEMU 实现的 VM 上来使用 RBD 块设备,那么 Linux Kernel 中的块设备驱动是 `virtio_blk`,它将对块设备的各种请求封装成一个消息通过 `virtio` 框架提供的队列发送到 QEMU 的 I/O 线程,QEMU 收到请求后会转给相应的 QEMU Block Driver 来完成请求。QEMU 作为最终使用 Librbd 中 RBDCache 的用户,它在 VM 关闭、QEMU 支持的热迁移操作或者 RBD 块设备卸载时也都会调用 QEMU Block Driver 的 Flush 接口,确保数据不会被丢失。

因此,需要用户在使用了开启 RBDCache 的 RBD 块设备时(此时 QEMU Block Driver 是 RBD,缓存也就是 RBDCache 就会交给 Librbd 自身去维护),需要给 QEMU 传入“`cache=writeback`”确保 QEMU 知晓有缓存的存在,不然 QEMU 会认为后端并没有缓存而选择将 Flush Request 忽略。

综上所述,可以发现开启 RBDCache 的 RBD 块设备实际上就是一个不带电容的磁盘,我们需要让文件系统开启 `barrier` 模式,幸运的是,这也是文件系统的默认情况。除此之外,因为文件系统实际上可能管理的是通过 LVM 这种逻辑卷管理工具得到的分区,因此必须确保文件系统下面的 Linux Device Mapping 层(一种从逻辑设备到物理设备的映射框架机制,在该机制下,用户可以很方便地根据自己的需要制定实现存储资源的管理策略,LVM2 基于该机制实现)也能够支持 Flush Request,LVM 在较早版本的 Kernel 中就已经支持 Flush Request,而其他 DM 模块可能就会忽略该请求,这就需要用户非常明确的了解。

幸运的是,RBD 会默认开启一个叫“`rbd_cache_writethrough_until_flush`”的选项,它的作用就是为了避免一些不支持“Flush”的 VM 来使用 RBDCache,它的主要方式是在用户开启 RBDCache 的情况下,在收到来自 VM 的第一个 Flush 请求前,它是不会在逻辑上启用 Cache 的。这样就避免了旧内核不支持 Flush 的问题。(参见麦子迈 <http://www.wzxue.com/rbdcache/>)

6.4.4 Ceph FS

Ceph 文件系统(Ceph FS)提供与 POSIX 兼容的文件系统服务,它使用 Ceph 存储集群

RADOS 来存储数据，Ceph FS 内的文件被映射到 RADOS 内的对象。

如图 6-36 所示，有两种使用 Ceph FS 的方式：一是通过 Kernel Module，Linux 内核里包含 Ceph FS 的实现代码；二是通过 FUSE（用户空间文件系统）的方式，通过调用 libcephfs 库来实现 Ceph FS 的加载，而 libcephfs 库又调用 librados 库与 RADOS 进行通信。

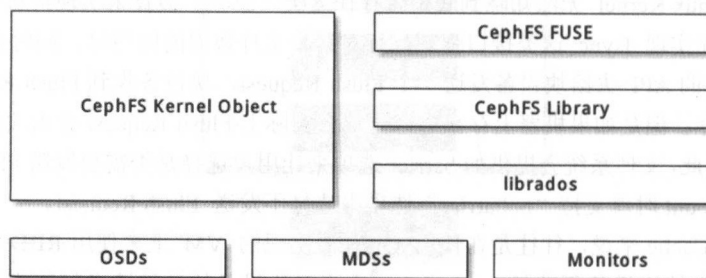


图 6-36 Ceph 文件系统

Ceph FS 要求 Ceph 存储集群内至少有一个元数据服务器（MDS），负责提供文件系统元数据（目录、文件所有者、访问模式等）的存储与操作。MDS 只为 Ceph FS 服务，如果不需要使用 Ceph FS，不需要配置 MDS。

Ceph FS 从数据中分离出了元数据，并存储于 MDS，文件数据存储于存储集群中的一或多个对象。MDS（名为 `ceph-mds` 的守护进程）存在的原因是，简单的文件系统操作像列出目录（`ls`）、或进入目录（`cd`）这些操作会不必要地扰动 OSD，所以把元数据从数据里分离出来意味着 Ceph 文件系统既能提供高性能服务，又能减轻存储集群负载。

6.4.5 Ceph 与 OpenStack

Radhat 架构师 Keith Tenzer 在最近的一篇文章（<http://superuser.openstack.org/articles/ceph-as-storage-for-openstack/>）讨论了如何将 Ceph 与 OpenStack 集成，以及为什么 Ceph 非常适合 OpenStack。

Ceph 提供统一的分布式存储服务，能够基于带有自我修复和智能预测故障功能的商用 x86 硬件进行横向扩展。它已经成为软件定义存储事实上的标准。因为 Ceph 是开源的，它使许多供应商能够提供基于 Ceph 的软件定义存储系统。Ceph 不仅限于 Red Hat、Suse、Mirantis、Ubuntu 等公司，SanDisk、富士通、惠普、戴尔、三星等公司现在也提供集成解决方案，甚至还有大规模的由社区构建的环境（如 CERN，即欧洲核子研究委员会），为上万个虚拟机提供存储服务。

Ceph 绝不局限于 OpenStack，这正是 Ceph 开始越来越受欢迎的原因。近期的 OpenStack 用户调查显示，Ceph 是 OpenStack 存储领域的显著领导者。2016 年 4 月 OpenStack 用户调查报告的第 42 页显示，Ceph 占 OpenStack 存储的 57%，下一个是 LVM（本地存储）占 28%，

NetApp 占 9%。如果我们不看 LVM, Ceph 领先其他存储公司 48%。

产生这种局面的原因有很多, 最重要的有 3 个:

- Ceph 是一个横向扩展的统一存储平台。OpenStack 最需要的存储能力有两个方面: 能够与 OpenStack 本身一起扩展, 并且扩展时不需要考虑是块(Cinder)、文件(Manila)还是对象(Swift)。传统存储供应商需要提供两个或 3 个不同的存储系统来实现这一点。
- Ceph 具有成本效益。Ceph 利用 Linux 作为操作系统, 而不是专有的系统。用户不仅可以选择找谁购买 Ceph 服务, 还可以选择从哪里购买硬件, 可以是同一供应商也可以是不同的。用户可以购买硬件, 甚至从单一供应商购买 Ceph + 硬件的集成解决方案。
- 和 OpenStack 一样, Ceph 也是开源项目, 这允许更紧密的集成和跨项目开发。

图 6-37 显示了所有需要存储的不同 OpenStack 组件, Ceph 如何与它们集成, 以及 Ceph 如何通过一个统一的存储系统满足所有的用例。

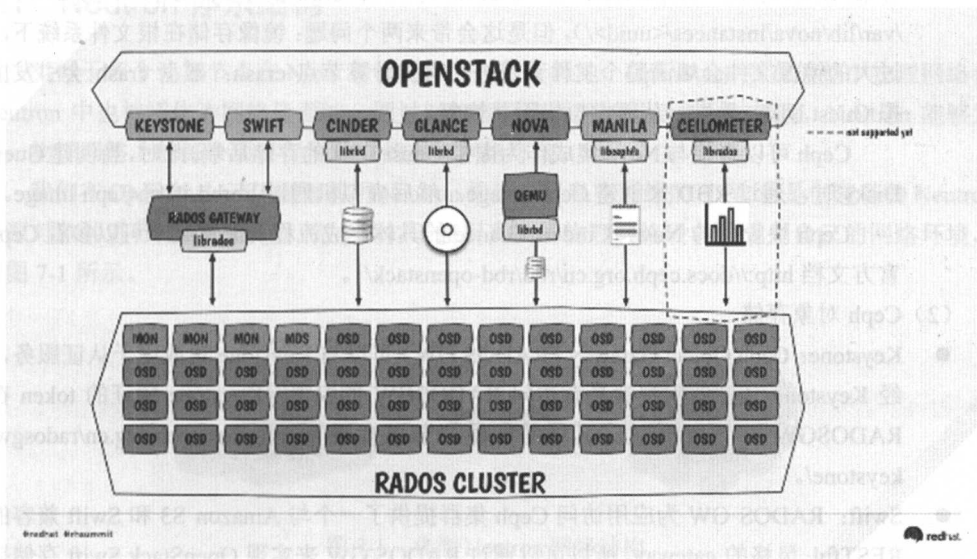


图 6-37 Ceph 与 OpenStack 集成

(1) Ceph 块存储

OpenStack 里有 3 个地方可以和 Ceph 块设备结合。

- Glance: Glance 是 OpenStack 中的镜像服务。默认情况下, 镜像存储在本地, 然后在被请求时复制到计算节点, 计算节点缓存镜像, 但每次更新镜像时, 都需要再次复制。

Ceph 可以为 Glance 提供存储后端, 允许镜像存储在 Ceph 中, 而不是存储在控

制节点和计算节点上。这大大减少了抓取镜像时的网络流量，提高了性能。此外，它使不同 OpenStack 部署之间的迁移变得更简单。

- **Cinder:** Cinder 是 OpenStack 中的块存储服务。Cinder 提供了关于块存储的抽象，并允许供应商通过提供驱动程序进行集成。在 Ceph 中，每个存储池可以映射到不同的 Cinder 后端，这允许创建诸如金、银或铜的存储服务，用户可以决定金是三副本的快速 SSD 磁盘，银是二副本的，铜则是使用 EC 纠删码的慢速磁盘。

Ceph 既可以配置为 Cinder 的存储后端，来提供虚机的快存储，也可以作为 Cinder 的备份存储后端，用来备份 Cinder 的卷数据。

```
$Cinder.conf
```

```
volume_driver = cinder.volume.drivers.rbd.RBDDriver
backup_driver= cinder.backup.drivers.ceph
```

- **Nova:** 默认情况下，Nova 将虚拟 Guest Disks（装有客户操作系统的磁盘）的镜像存储在本地的 Hypervisor 上（表现为文件系统的一个文件，通常位于 `/var/lib/nova/instances/<uuid>/`），但是这会带来两个问题：镜像存储在根文件系统下，过大的镜像文件会填满整个文件系统进而引发计算节点 crash；磁盘 crash 会引发虚拟 Guest Disks 丢失，从而虚拟机无从恢复。

Ceph 可以直接与 Nova 集成作为虚拟 Guest Disk 的存储后端。此时，当创建 Guest Disks 时，通过 RBD 来创建 Ceph Image，然后虚拟机通过 librbid 访问 Ceph Image。

Ceph 块设备与 Nova、Cinder、Glance 的具体集成流程及配置细节可以参看 Ceph 官方文档 <http://docs.ceph.org.cn/rbd/rbd-openstack/>。

(2) Ceph 对象存储

- **Keystone:** Ceph Object Gateway (RADOS GW) 可以与 Keystone 集成用于认证服务，经 Keystone 认证的用户就具有访问 RADOSGW 的权限，Keystone 验证的 token 在 RADOSGW 中同样有效。具体的配置细节可参看 <http://docs.ceph.org.cn/radosgw/keystone/>。
- **Swift:** RADOS GW 为应用访问 Ceph 集群提供了一个与 Amazon S3 和 Swift 兼容的 RESTful 风格的 gateway，所以可以通过 RADOS GW 来实现 OpenStack Swift 存储接口。对于 Ceph 与 Swift 的对比这里不做赘述。

(3) Ceph 文件系统

目前 OpenStack 中能够与 Ceph FS 集成的项目为 Manila (File Share Service)，OpenStack Manila 项目从 2013 年 8 月份开始进入社区视野，主要由 EMC、NetApp 和 IBM 的开发者驱动，是一个提供文件共享服务 API 并封装不同后端存储驱动的 Big Tent 项目。目前 Manila 中已经有 Ceph FS 驱动的实现。

与存储一样，网络也是 OpenStack 所管理的最重要的资源之一。Nova 实现了 OpenStack 虚拟机世界的抽象，Swift 与 Cinder 为虚拟机提供了安身之本，但是没有网络，任何虚拟机都将只是这个世界中的孤岛，不知道自己生存的价值。

最初，OpenStack 中的网络服务由 Nova 中一个单独的模块 nova-network 来提供，但是为了提供更为丰富的拓扑结构，支持更多的网络类型，具有更好的可扩展性，一个专门的项目 Neutron 被创建用于取代原有的 nova-network。

7.1 Neutron 体系结构

类似于各个计算节点在 Nova 中被泛化为计算资源池，OpenStack 所在的整个物理网络在 Neutron 中也被泛化为网络资源池，通过对物理网络资源的灵活划分与管理，Neutron 能够为同一物理网络上的每个项目提供独立的虚拟网络环境。

我们在 OpenStack 云环境里基于 Neutron 构建自己私有网络的过程，就是创建各种 Neutron 资源对象并进行连接的过程，完全类似于使用真实的物理网络设备来规划自己的网络环境，如图 7-1 所示。

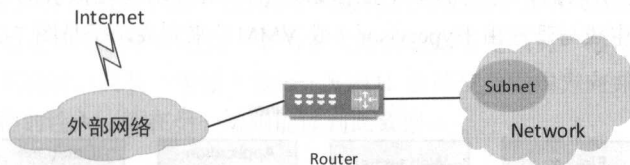


图 7-1 典型 Neutron 网络结构

首先，应该至少有一个由管理员所创建的外部网络对象来负责 OpenStack 环境与 Internet 的连接，然后项目可以创建自己私有的内部网络并在其中创建虚拟机，为了使内部网络中的虚拟机能够访问互联网，必须创建一个路由器将内部网络连接到外部网络，具体可参考使用 Horizon 来创建网络的过程。

这个过程中，Neutron 提供了一个 L3(三层)的抽象 router 与一个 L2(二层)的抽象 network，router 对应于真实网络环境中的路由器，为用户提供路由、NAT 等服务，network 则对应于一个真实物理网络中的二层局域网 (LAN)，从项目的角度看，它为项目所私有。

这里的 subnet 从 Neutron 的实现上来看并不能完全理解为物理网络中的子网概念。subnet 属于网路中的 3 层概念，指定一段 IPv4 或 IPv6 地址并描述其相关的配置信息，它附加在一个二层 network 上指明属于这个 network 的虚拟机可使用的 IP 地址范围。一个 network 可以同时拥有一个 IPv4 subnet 和一个 IPv6 subnet，除此之外，即使我们为其配置多个 subnet，也并不能够工作，可参考 <https://bugs.launchpad.net/neutron/+bug/1324459> 上的 bug 描述。

7.1.1 Linux 虚拟网络

Neutron 最为核心的工作是对二层物理网络 network 的抽象与管理。在一个传统的物理网络里，可能有一组物理的 Server，上面分别运行有各种各样的应用，比如 Web 服务、数据库服务等。为了彼此之间能够互相通信，每个物理 Server 都拥有一个或多个物理网卡（NIC），这些 NIC 被连接在物理交换设备上，比如交换机（Switch），如图 7-2 所示。

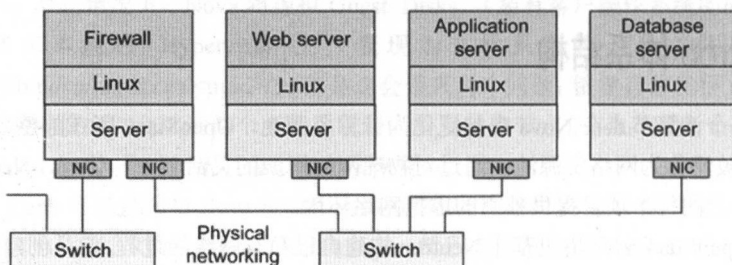


图 7-2 传统二层物理网络

虚拟化技术被引入后，上述的多个操作系统和应用可以以虚拟机的形式分享同一物理 Server，虚拟机的生成与管理由 Hypervisor（或 VMM）来完成，于是图 7-2 所示的网络结构被演化为图 7-3。

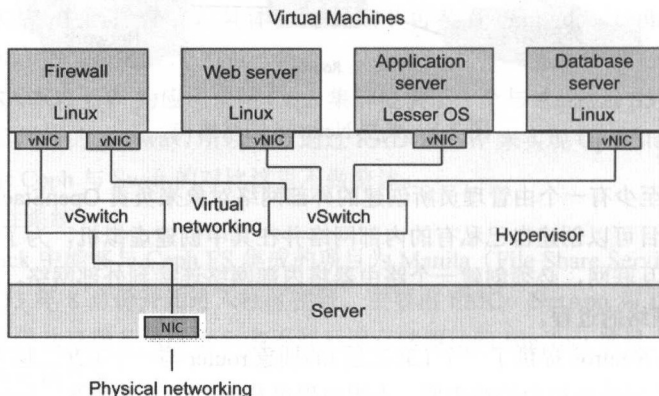


图 7-3 虚拟网络结构

虚拟机的网络功能由虚拟网卡 (vNIC) 提供, Hypervisor 可以为每个虚拟机创建一个或多个 vNIC, 站在虚拟机的角度, 这些 vNIC 等同于物理的网卡。为了实现与传统物理网络等同的网络结构, 与 NIC 一样 Switch 也被虚拟化为虚拟交换机 (vSwitch), 各个 vNIC 连接在 vSwitch 的端口上, 最后这些 vSwitch 通过物理 Server 的物理网卡访问外部的物理网络。

由此可见, 对一个虚拟的二层网络结构来说, 主要是完成两种网络设备的虚拟化: NIC 硬件与交换设备。Linux 环境下网络设备的虚拟化主要有以下几种形式, Neutron 也是基于这些技术来完成项目私有虚拟网络 network 的构建。

(1) TAP/TUN/VETH

TAP/TUN 是 Linux 内核实现的一对虚拟网络设备, TAP 工作在二层, TUN 工作在三层, Linux 内核通过 TAP/TUN 设备向绑定该设备的用户空间程序发送数据, 反之, 用户空间程序也可以像操作硬件网络设备那样, 通过 TAP/TUN 设备发送数据。

基于 TAP 驱动, 即可以实现虚拟网卡的功能, 虚拟机的每个 vNIC 都与 Hypervisor 中的一个 TAP 设备相连。当一个 TAP 设备被创建时, 在 Linux 设备文件目录下将会生成一个对应的字符设备文件, 用户程序可以像打开普通文件一样打开这个文件进行读写。

当对这个 TAP 设备文件执行 write() 操作时, 对于 Linux 网络子系统来说, 就相当于 TAP 设备收到了数据, 并请求内核接受它, Linux 内核收到此数据后将根据网络配置进行后续处理, 处理过程类似于普通的物理网卡从外界收到数据。当用户程序执行 read() 请求时, 相当于向内核查询 TAP 设备上是否有数据需要被发送, 有的话则取出到用户程序里, 从而完成 TAP 设备发送数据的功能。在这个过程中, TAP 设备可以被当做是本机的一个网卡, 而操作 TAP 设备的应用程序相当于另外一台计算机, 它通过 read/write 系统调用, 和本机进行网络通信, Subnet 属于网路中的 3 层概念, 指定一段 IPv4 或 IPv6 地址及并描述其相关的配置信息, 它附加在一个二层 network 上, 并指明属于这个 network 的虚拟机可使用的 IP 地址范围。

VETH 设备总是成对出现, 送到一端请求发送的数据总是从另一端以请求接受的形式出现。创建并配置正确后, 向其一端输入数据, VETH 会改变数据的方向并将其送入内核网络子系统, 完成数据的注入, 而在另一端则能读到此数据。

(2) Linux Bridge

Linux Bridge (网桥) 是工作于二层的虚拟网络设备, 功能类似于物理的交换机。

Bridge 可以绑定其他 Linux 网络设备作为从设备, 并将这些从设备虚拟化为端口, 当一个从设备被绑定到 Bridge 上时, 就相当于真实网络中的交换机端口插入了一个连接有终端的网线。

如图 7-4 所示, Bridge 设备 br0 绑定了实际设备 eth0 与虚拟设备 tap0/tap1, 此时, 对于 Hypervisor 的网络协议栈上层来说, 只看得到 br0, 并不会关心桥接的细节。当这些从设备接收到数据包时, 会将其提交给 br0 决定数据包的去向, br0 会根据 MAC 地址与端口的映射关系进行转发。

因为 Bridge 工作在第二层, 所以绑定在 br0 上的从设备 eth0、tap0 与 tap1 均不需要再设

置 IP，对上层路由器来说，它们都位于同一子网，因此只需为 br0 设置 IP（Bridge 设备虽然工作于二层，但它只是 Linux 网络设备抽象的一种，能够设置 IP 也可以理解），比如 10.0.1.0/24。此时，eth0、tap0 与 tap1 均通过 br0 处于 10.0.1.0/24 网段。

因为具有自己的 IP，br0 可以被加入到路由表，并利用它来发送数据，而最终实际的发送过程则由某个从设备来完成。

如果 eth0 本来具有自己的 IP，比如 192.168.1.1，在绑定到 br0 上之后，它的 IP 即会失效，用户程序不能接收到发送到这个 IP 的数据。只有目的地址为 br0 IP 的数据包才会被 Linux 接收。

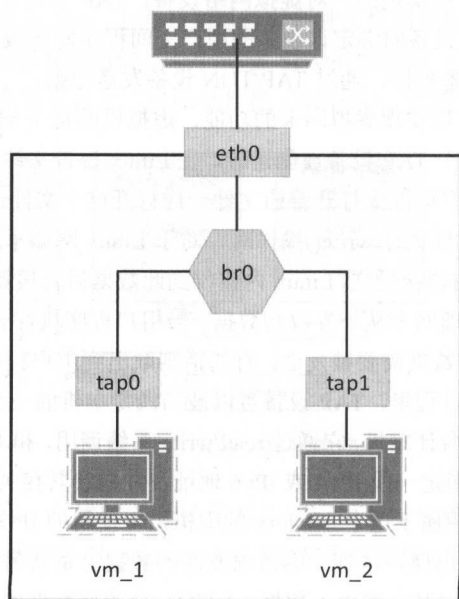


图 7-4 Linux Bridge 结构

(3) Open vSwitch

Open vSwitch (<http://openvswitch.org/>) 是一个具有产品级质量的虚拟交换机，它使用 C 语言进行开发，从而充分考虑了在不同虚拟化平台间的移植性，同时，它遵循 Apache 2.0 许可，因此对商用也非常友好。

如前所述，对于虚拟网络来说，交换设备的虚拟化是很关键的一环，vSwitch 负责连接 vNIC 与物理网卡，同时也桥接同一物理 Server 内的各个 vNIC。Linux Bridge 已经能够很好地充当这样的角色，为什么我们还需要 Open vSwitch？

Open vSwitch 在文章 WHY-OVS 中首先高度赞扬了 Linux Bridge 之后，给出了详细的解答：

We love the existing network stack in Linux. It is robust, flexible, and feature rich. Linux already contains an in-kernel L2 switch (the Linux bridge) which can be used by VMs for inter-VM communication. So, it is reasonable to ask why there is a need for a new network switch.

在传统数据中心的网络管理中，网络管理员通过对交换机的端口进行一定的配置，可以很好地控制物理机的网络接入，完成网络隔离、流量监控、数据包分析、Qos 配置、流量优化等一系列工作。

但是在云环境中，仅凭物理交换机的支持，管理员无法区分被桥接的物理网卡上流淌的数据包属于哪个 VM、哪个 OS 及哪个用户，Open vSwitch 的引入则使云环境中虚拟网络的管理以及对网络状态和流量的监控变得容易。

比如，我们可以像配置物理交换机一样，将接入到 Open vSwitch（Open vSwitch 同样会在物理 Server 上创建一个或多个 vSwitch 供各个虚拟机接入）上的各个 VM 分配到不同的 VLAN 中实现网络的隔离。我们也可以在 Open vSwitch 端口上为 VM 配置 Qos，同时 Open vSwitch 也支持包括 NetFlow、sFlow 很多标准的管理接口和协议，我们可以通过这些接口完成流量监控等工作。

此外，Open vSwitch 也提供了对 Open Flow 的支持，可以接受 Open Flow Controller 的管理。

总之，Open vSwitch 在云环境中的各种虚拟化平台上（比如 Xen 与 KVM）实现了分布式的虚拟交换机（Distributed Virtual Switch），一个物理 Server 上的 vSwitch 可以透明地与另一 Server 上的 vSwitch 连接在一起，如图 7-5 所示。

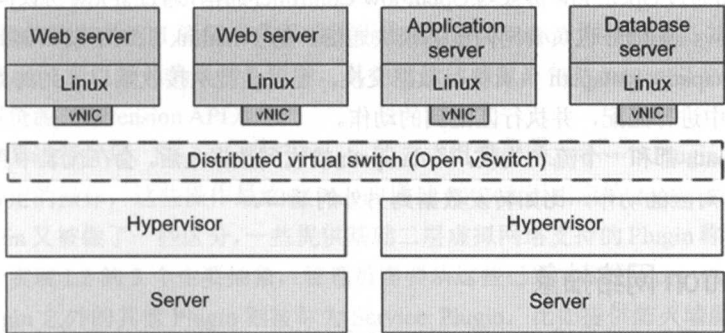


图 7-5 Open vSwitch

而至于 Open vSwitch 软件本身，则由内核态的模块以及用户态的一系列后台程序所组成，结构如图 7-6 所示。

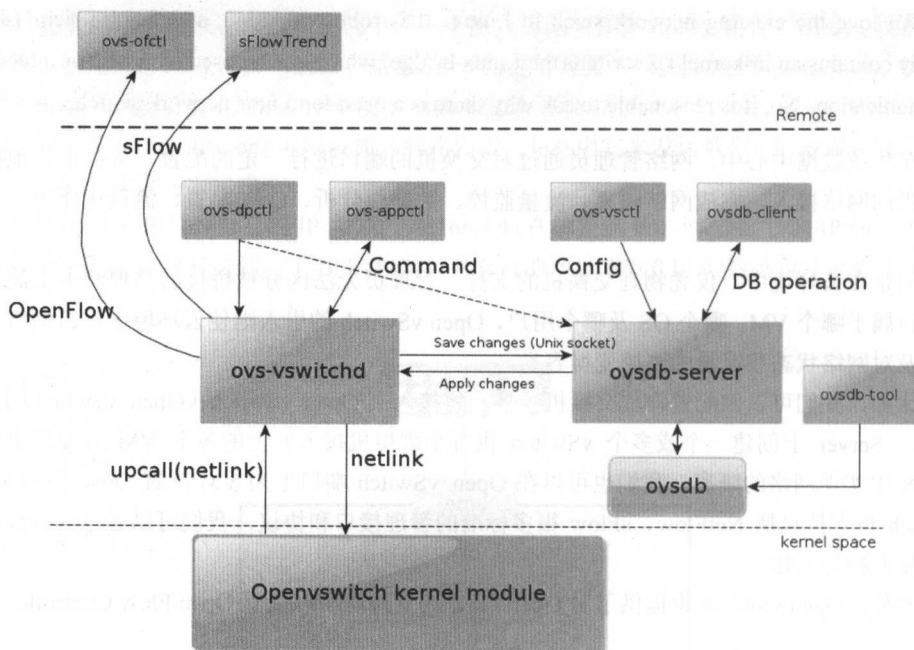


图 7-6 Open vSwitch 软件结构

其中 ovs-vswitchd 是最主要的模块，实现了虚拟机交换机的后台，负责同远程的 Controller 进行通信，比如通过 OpenFlow 协议与 OpenFlow Controller 通信，通过 sFlow 协议同 sFlow Trend 通信。此外，ovs-switchd 也负责同内核态模块通信，基于 netlink 机制下发具体的规则和动作到内核态的 datapath，datapath 负责执行数据交换，也就是把从接收端口收到的数据包在流表（Flow Table）中进行匹配，并执行匹配到的动作。

每个 datapath 都和一个流表关联，当 datapath 接收到数据之后，会在流表中查找可以匹配的 Flow，执行对应的动作，比如转发数据到另外的端口。

7.1.2 Neutron 网络抽象

目前为止，我们已经知道 Neutron 通过 L3 的抽象 router 提供路由器的功能，通过 L2 的抽象 network/subnet 完成对真实二层物理网络的映射，并且 network 有 Linux Bridge、Open vSwitch 等不同的实现方式。

除此之外，在 L2 中，Neutron 还提供了一个重要的抽象 port，代表了虚拟交换机上的一个虚拟交换端口，记录其属于哪个网络以及对应的 IP 等信息。当一个 port 被创建时，默认情况下，会为它分配其指定 subnet 中可用的 IP。当我们创建虚拟机时，可以为其指定一个 port。

对于 L2 层抽象 network 来说，必然需要映射到真正的物理网络，但 Linux Bridge 与 Open

vSwitch 等只是虚拟网络的底层实现机制，并不能代表物理网络的拓扑类型。目前 Neutron 主要实现了以下几种网络类型的支持。

- Flat: Flat 类型的网络不支持 VLAN，因此不支持二层隔离，所有虚拟机都在一个广播域。
- VLAN: 与 Flat 相比，VLAN 类型的网络自然会提供 VLAN 的支持。
- NVGRE: NVGRE (Network Virtualization using Generic Routing Encapsulation) 是点对点的 IP 隧道技术，可以用于虚拟网络互联。NVGRE 容许在 GRE 内传输以太网帧，而 GRE key 拆成两部分，前 24 位作为 Tenant ID，后 8 位作为 Entropy 用于区分隧道两端连接的不同虚拟网络。
- VxLAN: Vxlan (Virtual Extensible LAN) 技术的本质是将 L2 层的数据帧头重新定义后通过 L4 层的 UDP 进行传输。相较于采用物理 VLAN 实现的网络虚拟化，VxLAN 是 UDP 隧道，可以穿越 IP 网络，使得两个虚拟 VLAN 可以实现二层联通，并且突破 4095 的 VLAN ID 限制提供多达 1600 万的虚拟网络容量。

除了上述 L2 与 L3 的抽象，Neutron 还提供了更高层次的一些服务，主要有 FWaaS、LBaaS 和 VPNaaS。

7.1.3 Neutron 架构

不同于 Nova 与 Swift，Neutron 只有一个主要的服务进程 `neutron-server`，它运行于网络控制节点上，提供 RESTful API 作为访问 Neutron 的入口，`neutron-server` 接收到的用户 HTTP 请求最终由遍布于计算节点和网络节点上的各种 Agent 来完成。

Neutron 提供的众多 API 资源对应了前面所述的各种 Neutron 网络抽象，其中 L2 的抽象 `network/subnet/port` 可以被认为是核心资源，其他层次的抽象，包括 `router` 以及众多的高层次服务则是扩展资源 (Extension API)。

为了更容易进行扩展，Neutron 利用 Plugin 的方式组织代码，每一个 Plugin 支持一组 API 资源并完成特定的操作，这些操作最终由 Plugin 通过 RPC 调用相应的 Agent 来完成。

这些 Plugin 又被做了一些区分，一些提供基础二层虚拟网络支持的 Plugin 称为 Core Plugin，它们必须至少实现 L2 的 3 个主要抽象，管理员需要从这些已经实现的 Core Plugin 中选择一种。Core Plugin 之外的其他 Plugin 则被称为 Service Plugin，比如提供防火墙服务的 Firewall Plugin。

至于 L3 抽象 `router`，许多 Core plugin 并没有实现，H 版本之前他们是采用 Mixin 设计模式，将标准的 `router` 功能包含进来，以提供 L3 服务给项目。H 版本之中，Neutron 实现了一个专门的名为 L3 Router Service Plugin 提供 `router` 服务。

Agent 一般专属于某个功能，用于使用物理网络设备或一些虚拟化技术完成某些实际的操作。比如实现 `router` 具体操作的 L3 Agent。

在 K 版本以前，Neutron 代码中还包括了针对不同厂商的 Plugin，但是从 K 版本开始，

这些 Plugin 被从 Neutron 代码中移除，Neutron 社区不再负责维护这些厂商定制的 Plugin。同时被移除的还有在 ML2 框架下与厂商及其他开源 SDN 解决方案（比如 OpenDaylight、OVN 等）的相关 driver，这些 driver 被放置到独立的代码仓库中被相关社区维护（譬如 networking-odl、networking-ovn 等）。

Neutron 的完整架构如图 7-7 所示。

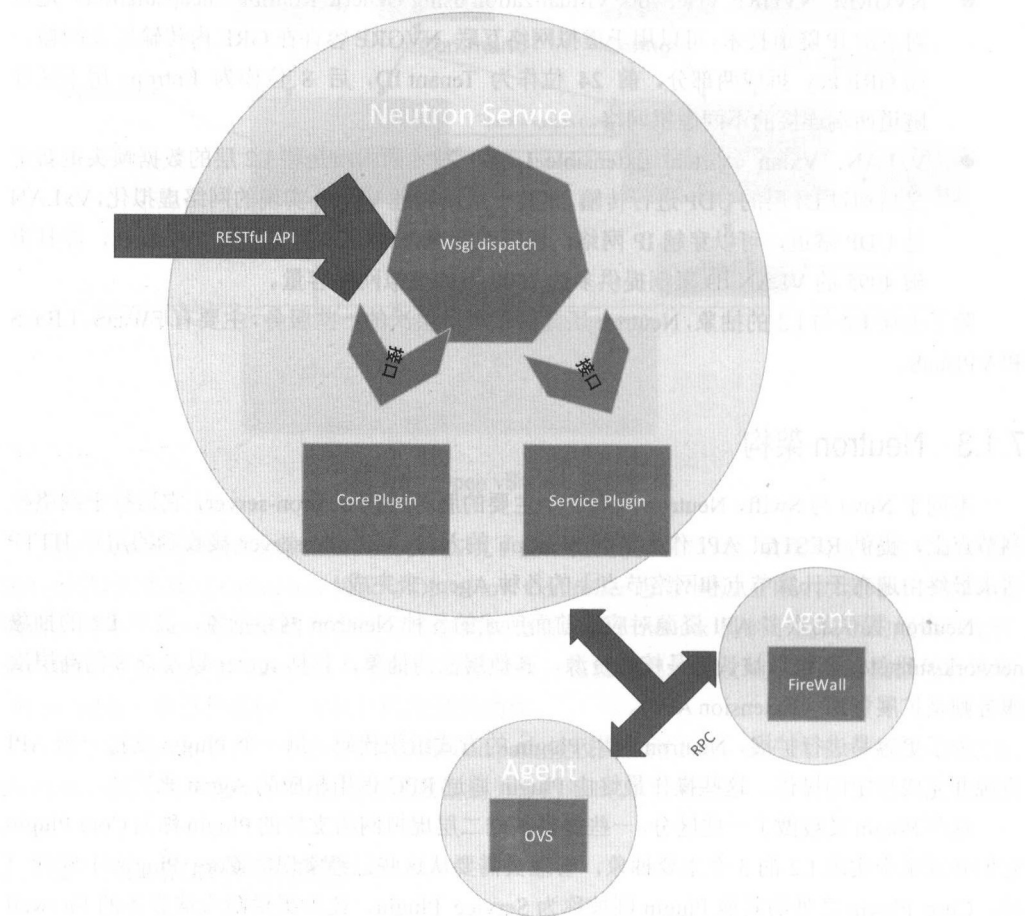


图 7-7 Neutron 架构

因为各种 Core Plugin 的实现之间存在很多重复的代码，比如对数据库的访问操作，所以 H 版本中 Neutron 实现了一个 ML2 Core Plugin，它采用了更加灵活的结构进行实现，通过 Driver 的形式可以对现有的各种 Core Plugin 提供支持，因此可以说 ML2 Plugin 的出现意在取代目前的所有 Core Plugin。

对于 ML2 Core Plugin 以及各种 Service Plugin 来说，虽然有被剥离出 Neutron 作为独立项

目存在的可能，但它们的基本实现方式与本章所涵盖的内容相比并不会发生大的改变。

7.1.4 Neutron 源码结构

Neutron 源码目录结构如下：

```
.
|-- api-ref/ - API reference 已经被转移到 neutron-lib 代码仓库中
|-- babel.cfg
|-- bin/
|   |-- neutron-rootwrap-xen-dom0 - 用于在 xen dom0 上以 root 权限执行命令
|-- devstack/ - devstack 插件
|-- doc/ - 用于文档生成
|-- etc/ - 生成各种配置文件，基本上都会在安装时被复制到/etc/目录下
|-- neutron/ - neutron 核心代码
|   |-- agent/ - 各种 agent，定义了各种 agent 的 RPC 接口，以及对 agent 常用
|               命令行工具（ip 命令、ovs 命令等等）的封装。但是 L2 agent
|               （LinuxBridge Agent, OpenvSwitch agent）等各种具体 agent
|               的实现位于 plugins/ml2/drivers/目录下
|   |-- api/ - Neutron API 的实现
|   |-- auth.py - 从 keystone 认证完成之后返回的 http header 构建 request context
|   |-- callbacks/ - Neutron 自己实现的 callback 机制
|   |-- cmd/ - 常用命令的封装
|   |-- common/ - 公共常量和函数的定义
|   |-- conf/ - 定义和处理各种配置选项
|   |-- context.py - 对 request context 的定义
|   |-- core_extensions/ - 对 core resource 的扩展
|   |-- db/ - 数据库数据模型以及 API
|   |-- debug/ - 辅助的调试工具
|   |-- extensions/ - 各种 Extension API 实现
|   |-- hacking/ - 编码规范检查
|   |-- ipam/ - IP 地址空间管理
|   |-- locale/ - 多语言支持
|   |-- manager.py - 解析配置文件并加载 Core Plugin
|   |-- neutron_plugin_base_v2.py - Neutron Plugin 的抽象基类
|   |-- notifiers/ - 通知 Nova 网络有关的变化
|   |-- objects/ - 为 rolling upgrade 准备的 oslo versioned objects
|   |-- opts.py - oslo config opts 的辅助函数
|   |-- pecan_wsgi/ - 基于 pecan 实现的 wsgi
|   |-- plugins/ - 各种 Core Plugin 实现
|   |-- policy.py - Neutron RBAC 支持
|   |-- quota/ - 用于注册、标记、追踪需要限制 quota 的资源
|   |-- scheduler/ - 调度器（DHCP、L3 agent 的调度）的实现
|   |-- server/ - Neutron server wsgi 入口
```

```
| |-- service.py - 定义各种服务的基类, 比如类 WsgiService 是 WSGI 服务的基类
| |-- services/ - 各种 Service Plugin 的实现
| |-- tests/ - 单元和功能测试
| |-- version.py
| |-- worker.py
| |-- wsgi.py - wsgi server 的工具性函数
|-- rally-jobs/ - Rally 相关测试场景和配置
|-- releasenotes/
|-- run_tests.sh
|-- setup.cfg
|-- setup.py
|-- tools/ -包括一些安装、格式检查等相关的工具
`-- tox.ini
```

依照惯例, 接下来我们首先浏览 `setup.cfg` 文件。

```
console_scripts =
    neutron-db-manage = neutron.db.migration.cli:main
    neutron-debug = neutron.debug.shell:main
    neutron-dhcp-agent = neutron.cmd.eventlet.agents.dhcp:main
    neutron-l3-agent = neutron.cmd.eventlet.agents.l3:main
    .....

neutron.core_plugins =
    ml2 = neutron.plugins.ml2.plugin:Ml2Plugin

neutron.service_plugins =
    dummy = neutron.tests.unit.dummy_plugin:DummyServicePlugin
    router = neutron.services.l3_router.l3_router_plugin:L3RouterPlugin
    metering = neutron.services.metering.metering_plugin:MeteringPlugin
    qos = neutron.services.qos.qos_plugin:QoSPlugin
    .....

neutron.ml2.type_drivers =
    flat = neutron.plugins.ml2.drivers.type_flat:FlatTypeDriver
    local = neutron.plugins.ml2.drivers.type_local:LocalTypeDriver
    vlan = neutron.plugins.ml2.drivers.type_vlan:VlanTypeDriver
    .....
```

命名空间“`neutron.core_plugins`”指明了 Core Plugin 的入口。目前 Neutron 的 Core Plugin 只有 ML2 Plugin, 之前版本中厂商定制的各种 Plugin 已经从 Neutron 代码中移除。

`neutron.service_plugins` 指明了各种 Service Plugin 实现的入口。Neutron 的很多新特性都是通过 Service Plugin 的方式来实现的, 比如 QoS、vlan-aware-vms 等。

`neutron.ml2.type_drivers` 和 `neutron.ml2.mechanism_drivers` 分别指明了在 ML2 框架下的

type driver 和 mechanism driver 的入口函数。

neutron.ml2.extension_drivers 指明了 ML2 Plugin 所实现的 Neutron API extension 的驱动入口。这些 extension driver 实现了针对 Neutron 核心资源 (port、subnet、network) 的功能的扩展。

neutron.ipam_drivers 指明了 Neutron 的 IP 地址管理 (IP address management) 的驱动入口。

neutron.agent.l2.extensions 指明了对原有 Neutron L2 Agent 的接口的扩展。这些扩展包括 QoS 在 L2 Agent 上实现的接口以及用于 SRIOV 的 FDB population 的接口。

neutron.qos.agent_drivers 指明了 QoS agent driver 的入口。

neutron.interface_drivers 指明了 Neutron 对各种二层网桥 (OVS、LinuxBridge 等) 的驱动程序的入口。

neutron.agent.firewall_drivers 指明了各种 firewall driver 的入口, 主要包括基于 iptables 的实现和 Open vSwitch 的 flow table 的实现。

console_scripts 这个命名空间则指明了 neutron-server 服务以及各种 Agent 实现的入口。此外, 还包括了一些辅助的命令或工具, 这些工具脚本在部署时会被安装。

- **neutron-db-manage**: 负责从旧 Neutron 版本的数据库迁移与升级到新的版本, 比如将原先的 Open vSwitch 或 Linux Bridge 的数据库迁移到 ML2 支持的数据库格式。具体可参考 neutron/db/migration/README 文件。
- **neutron-debug**: Neutron 辅助 Debug 脚本, 提供了一个 Shell 环境来进行 Debug, 具体可参考 neutron/debug/README 文件。
- **neutron-netns-cleanup**: 清理无用的 Network Namespace, 用于当 Neutron 的 Agent 异常退出时进行环境的清理工作。Network Namespace 的引入是为了支持网络协议栈的多个运行实例从而实现网络的隔离, 类似于进程的线性地址空间。
- **neutron-ovs-cleanup**: 清理无用的 Open vSwitch 网桥 (或者说 vSwitch) 和端口。
- **neutron-sanity-check**: 执行一些简单的检查, 比如是否支持 VxLAN 等。

7.2 Neutron API

类似于 Nova 对 API 资源的管理方式, Neutron 也将基于各种网络抽象得到的 API 资源分为核心资源 (Core API) 与扩展资源 (Extension API) 两种, Core API 只对应 L2 层的 network/subnet/port 3 种抽象, 其余的各层抽象都涵盖在 Extension API 的范围。

Neutron API 实现的主要代码位于 neutron/api 目录:

```
.
|-- api_common.py - 一些 API 实现通用的类与方法
|-- extensions.py - Neutron API extension 接口定义
|-- rpc
|   |-- agentnotifiers - 非核心资源的 Agent (L3、DHCP 等) 的 RPC 接口定义
```

```
| |-- callbacks - Neutron 实现的注册回调机制
| |-- handlers - Plugin 对 Agent 的 RPC 接口（非核心资源）
|-- v2
| |-- attributes.py - 核心资源的 API 属性定义及相关函数
| |-- base.py - WSGI controller 的基类
| |-- resource.py - WSGI resource 的基类
| |-- resource_helper.py - Resource 的辅助性函数
| |-- router.py - API router
|-- versions.py - 用户请求是版本号时，使用这个模块进行处理
`-- views
```

v2 目录包括了几个 L2 层核心资源的实现代码。依照 OpenStack 各个项目 API 的实现惯例，每个 Neutron API 资源都会被封装成一个 WSGI Application，Neutron API 服务进程 neutron-server 接收到用户的 HTTP 请求后会通过 Router 模块将其路由到相关资源的 Controller 中去执行对应操作。

但是我们并没有在 v2 目录下发现形如 network.py、subnet.py、port.py 的文件去实现这几个核心资源，更不会发现 NetworkController/SubnetController/PortController 这样的 Controller 类去处理具体的用户请求。

这是因为对于这 3 个核心资源来说，都使用了 base.py 文件中的类 Controller 去实现，只是在封装成 WSGI Application 的时候，调用这个文件中的 create_resource() 函数根据不同的参数动态创建对应的 Controller 对象。

```
def create_resource(collection, resource, plugin, params,
allow_bulk=False,
                    member_actions=None, parent=None, allow_pagination=False,
                    allow_sorting=False):
    controller = Controller(plugin, collection, resource, params, allow_bulk,
                           member_actions=member_actions, parent=parent,
                           allow_pagination=allow_pagination,
                           allow_sorting=allow_sorting)

    return wsgi_resource.Resource(controller, FAULT_MAP)
```

而对于其余的扩展资源，Neutron 仍然使用了传统的方式来实现，它们在 neutron/extensions/目录下都分别有对应的实现文件与对应的 Controller 类。位于 neutron/api/目录下的 extension.py 文件只是一些基类和公用的代码。

7.2.1 neutron-server

作为 Neutron 中的唯一一个服务进程，neutron-server 承担着接收用户 RESTful API 请求并分发处理的任务。

根据 setup.cfg 中的配置，neutron-server 的入口为 neutron.cmd.eventlet.server:main:


```
def main():
    server.boot_server(_main_neutron_server)

def _main_neutron_server():
    if cfg.CONF.web_framework == 'legacy':
        wsgi_eventlet.eventlet_wsgi_server()
    else:
        wsgi_pecan.pecan_wsgi_server()
```

Neutron 目前支持两套 WSGI 的实现，社区的意图是使用新的基于 Pecan 的实现来代替原有的实现。当“web_framework”被配置成“legacy”的时候就启用原有的基于 Paste 的实现，这个是目前默认的配置。

Paste Deploy 会在这个 WSGI Server 创建时参与进来，基于 Paste 配置文件/etc/nova/api-paste.ini 去加载 WSGI Application。

```
[composite:neutronapi_v2_0]
use = call:neutron.auth:pipeline_factory
noauth = request_id catch_errors extensions neutronapiapp_v2_0
keystone = request_id catch_errors authtoken keystonecontext extensions
neutronapiapp_v2_0

[app:neutronapiapp_v2_0]
paste.app_factory = neutron.api.v2.router:APIRouter.factory
```

APIRouter 类会完成所有 Core API 与 Extension API 的加载与路由规则的创建。从收到用户的 HTTP 请求到具体 Controller 中操作的过程与其他项目基本类似，可以参考 Nova 一章的描述。

Controller 中的操作具体执行时，会根据请求的资源 and Action，拼接出应该调用的接口交由 Plugin 处理。比如所请求操作的资源是 network，Action 是“CREATE”，则应该调用的 Plugin 接口即为“create_network”。

基于 Pecan 的实现在 neutron/server/wsgi_pecan.py 和 neutron/pecan_wsgi/目录下，有兴趣的读者可以自行阅读。

7.3 ML2 Plugin

H 版本中，ML2 Plugin 被添加意图取代所有的 Core Plugin，它采用了更加灵活的结构进行实现。图 7-8 所示即为 ML2 Plugin 的实现框架。

作为一个 Core Plugin，ML2 自然会实现 network/subnet/port 3 种核心资源，同时它也实现了包括 Port Binding 等在内的部分扩展资源。

ML2 解耦了网络拓扑类型与底层的虚拟网络实现机制，并分别通过 Driver 的形式进行扩展，其中，不同的网络拓扑类型对应着 Type Driver，由 Type Manager 管理，不同的网络实现

机制对应着 Mechanism Driver，由 Mechanism Manager 管理。

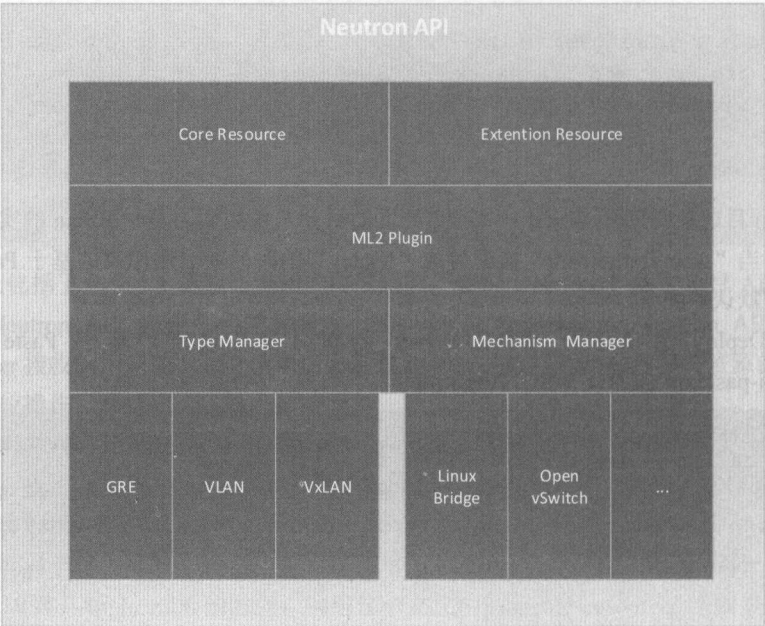


图 7-8 ML2 Plugin 框架

目前，Neutron 中已经实现了 Flat/GRE/VLAN/VxLAN 等拓扑类型的 Type Driver，也实现了 Linux Bridge/Open vSwitch 以及众多厂商的 Mechanism Driver。通过这些众多的 Driver，ML2 Plugin 实现了其他 Core Plugin 的功能。

ML2 Plugin 的源码结构如下：

```
.
├── common
├── config.py - 一些配置选项定义
├── db.py
├── driver_api.py - 定义 TypeDriver/MechanismDriver 基类
├── driver_context.py
├── drivers - 各种 TypeDriver/MechanismDriver 的实现
├── managers.py - 定义 TypeManager/MechanismManager 类
├── models.py
├── plugin.py - 类 Ml2Plugin 实现
└── rpc.py - 与 Agent 进行 RPC 交互
```

1. Ml2Plugin 类

```
class Ml2Plugin(db_base_plugin_v2.NeutronDbPluginV2,
```

```
dvr_mac_db.DVRDbMixin,
external_net_db.External_net_db_mixin,
sg_db_rpc.SecurityGroupServerRpcMixin,
agentschedulers_db.AZDhcpAgentSchedulerDbMixin,
addr_pair_db.AllowedAddressPairsMixin,
vlantransparent_db.Vlantransparent_db_mixin,
extradhcpopt_db.ExtraDhcpOptMixin,
address_scope_db.AddressScopeDbMixin,
service_type_db.SubnetServiceTypeMixin):
```

从 ML2Plugin 类的定义看，它通过继承众多 Mixin，能够支持许多的功能。由于具体设备的操作由 Agent 来完成，ML2 Plugin 本身实际上大都是在完成基于数据库的一些操作，致力于正确有效地管理 network/port/subnet 这些资源及其相互关系，同时正确地与 Agent 交互从而完成虚拟网络部署。ML2Plugin 类的众多基类中，除了 SecurityGroupServerRpcMixin 其他都是与数据库操作相关。

除了 3 种核心的资源外，ML2 也支持许多扩展的资源，ML2Plugin 类需要实现这些资源的操作接口，以供收到用户请求时资源对应的 Controller 来调用。ML2 支持的扩展资源在 ML2Plugin 类里有定义：

```
_supported_extension_aliases = ["provider", "external-net", "binding",
                                "quotas", "security-group", "agent",
                                "dhcp_agent_scheduler",
                                "multi-provider", "allowed-address-pairs",
                                "extra_dhcp_opt", "subnet_allocation",
                                "net-mtu", "vlan-transparent",
                                "address-scope",
                                "availability_zone",
                                "network_availability_zone",
                                "default-subnetpools",
                                "subnet-service-types"]
```

这些扩展资源里，并不是每一个的操作接口都由类 ML2Plugin 来实现，许多接口是由类 ML2Plugin 所继承的父类提供的。比如对于“security-group”，就由类 SecurityGroupServerRpcMixin 来提供操作的接口。

我们以创建一个名为 public0 的 Flat 类型的网络为例，network 创建无需通知 Agent，所有流程限于 ML2 内部。

执行 neutron 的客户端命令创建网络，并开启 debug 模式，这容许我们直接地看到传递给 Plugin 的各个参数的生成过程。

```
$ neutron net-create public01 --tenant_id 73ddc007555840f8b2ad72997cfe8ea6
--provider:network_type flat --provider:physical_network physnet1 --debug
```

通过 Router 模块的路由，这个命令最终由 Plugin 的 create_network()函数来完成：

```

@utils.transaction_guard
@db_api.retry_if_session_inactive()
def create_network(self, context, network):
    # 调用 create_network_db 函数完成 DB 条目的创建, 其中包括 type driver
    # 的相关接口调用, 以及 mechanism driver 的 precommit 函数的调用
    result, mech_context = self._create_network_db(context, network)
    kwargs = {'context': context, 'network': result}
    # 通知各种对 network 资源创建事件感兴趣的成员
    registry.notify(resources.NETWORK, events.AFTER_CREATE, self,
**kwargs)

    try:
        # mechanism driver 的 postcommit 函数
        self.mechanism_manager.create_network_postcommit(mech_context)
    except ml2_exc.MechanismDriverError:
        with excutils.save_and_reraise_exception():
            LOG.error(_LE("mechanism_manager.create_network_postcommit "
                           "failed, deleting network '%s'"), result['id'])
            self.delete_network(context, result['id'])

    return result

```

因为我们在命令执行时开启了 **debug** 模式, 可以很容易地获得字典 **network** 的内容:

```

DEBUG: keystoneclient.session REQ: curl -i -X POST
http://172.16.0.1:9696/v2.0/networks.json -H "User-Agent:
python-neutronclient" -H "Content-Type: application/json" -H "Accept:
application/json" -H "X-Auth-Token: TOKEN_REDACTED" -d '{"network":
{"tenant_id": "73ddc007555840f8b2ad72997cfe8ea6", "provider:network_type":
"flat", "name": "public01", "provider:physical_network": "physnet1",
"admin_state_up": true}}'

```

Type Manager 和 Mechanism Manager 的接口定义方式不同, 每个操作, Mechanism Manager 都有两个接口, 一个在数据库 session 内调用, 命名规则遵守 “{action}-{object}-precommit” 的形式; 另一个形如 “{action}-{object}-postcommit” 的接口则在数据库提交完成之后调用。而 Type Manager 的接口定义就没有这么规整, 比如上述代码中的 `create_network_segments()` 与 `_extend_network_dict_provider()`。

2. Provider Network 与 Multi-Segment Network

我们在上面创建 **network** 的命令与代码里已经看到 **provider** 与 **segment** 的字眼, 因为 ML2 中这两个概念对于虚拟网络与物理承载网络的映射非常重要, 这里有必要对其做专门的说明。

Segment 可以简单理解为对物理网络一部分的描述, 比如它可以是物理网络中很多 VLAN 中的一个 VLAN。ML2 仅仅使用下面的结构来定义一个 Segment:

```
{NETWORK_TYPE, PHYSICAL_NETWORK, and SEGMENTATION_ID}
```

如果 Segment 对应了物理网络中的一个 VLAN, 则这里的 SEGMENTATION_ID 就是这个 VLAN 的 VLAN ID; 如果 Segment 对应的是 GRE 网络中的一个 Tunnel, 则 SEGMENTATION_ID 就是这个 Tunnel 的 Tunnel ID。ML2 就是使用这样简单的方式将 Segment 与物理网络对应起来。

在 Neutron 还被称为 Quantum 的时代, 创建虚拟网络时不能指定 VLAN ID 或 Tunnel ID, 也就是说, 如果此时数据中心已经有了一个 VLAN 的 ID 为 100, 需要部署一些 VM 在这个 VLAN 上就比较困难。

当时的一些 Plugin, 比如 Linux Bridge 是可以做到这点的, 但是问题在于并没有一个统一的方法来达到这个目的, 所以提出了 Provider Network API 的需求, 经过一段时间的发展, 名为 Provider 的 Extension API 被添加来管理虚拟网络与物理承载网络之间的映射。换句话说, 所谓的 Provider Network 目的就是指创建虚拟网络时, Neutron 允许你指定这个虚拟网络所占用的物理网络资源。

2013 年初, 针对 Provider Extension API, 又提出了更进一步的改进需求, 允许将一个虚拟 network 与多个物理网络对应起来, 换句话说, 就是这个虚拟网络可以包含多个、多种不同的 Provider Network, 这也就是所谓的 Multi-Segment Network, 比如:

```
{
  "network": {
    "segments": [
      {
        "provider:segmentation_id": "2",
        "provider:physical_network":
"8bab8453-1bc9-45af-8c70-f83aa9b50453",
        "provider:network_type": "vlan"
      },
      {
        "provider:segmentation_id": "100",
        "provider:network_type": "gre"
      }
    ],
    "name": "net1",
    "admin_state_up": true
  }
}
```

Multi-Segment Network 网络能够灵活地使用现存物理网络作为承载网络, 当前有 ML2 和 NSX Plugin 对其提供了支持。考虑一个 Multi-Segment 的网络例子, 比如可以建立如图 7-9 所示的虚拟二层网络, 这个虚拟网络有两个现存的物理网 VLAN 5 和 VLAN 8 来承载。各个 Segment 中间的桥接是系统管理员的责任。

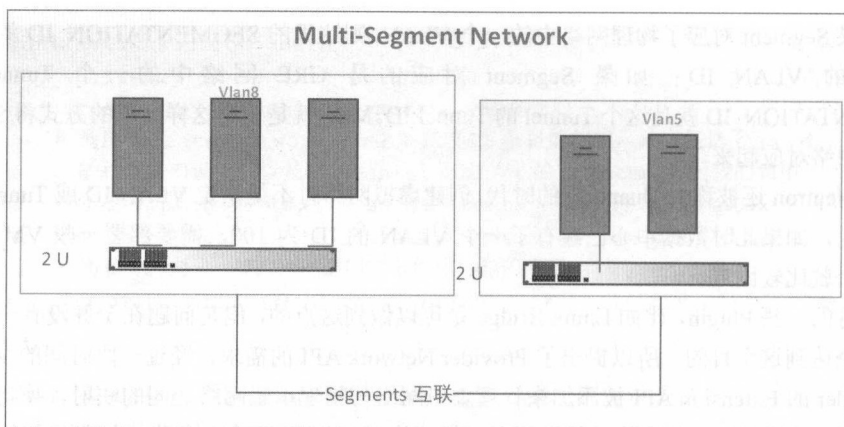


图 7-9 Multi-Segment Network

3. Type Manager 与 Mechanism Manager

Type manager 与 Mechanism Manager 负责加载对应的 Type Driver 和 Mechanism Driver，并将具体的操作分发到具体的 Driver。此外，一些 Driver 通用的代码也由 Manager 来提供。

Type Manager 在初始化的时候，会根据配置加载对应的 Type Driver。Type manager 与其管理的 Type Driver 一起提供了对 Segment 的各种操作，包括存储、验证、分配和回收等。创建一个 network 的时候需要从传递的参数中提取出 Segment 有关的信息，进行验证。

下面以创建一个 Flat 类型网络为例，命令中提供了 Segment 有关的全部信息：

```
$ neutron net-create public01 --tenant_id 73ddc007555840f8b2ad72997cfe8ea6
--provider:network_type vlan --provider:physical_network physnet1
--provider:segmentation_id 100 --debug
```

这种情况下的 Segment 称作 Provider Segment，Type manager 会从这个命令的参数中提取出相关信息构建一个 Segment 结构，然后告诉 Type Driver 保留这个 Provider Segment。

如果命令中没有提供这些信息：

```
$ neutron net-create tttt --tenant_id 73ddc007555840f8b2ad72997cfe8ea6
--debug
```

此时 Type manager 将通过 Type Driver 直接按需分配一个 Segment：

```
{'segmentation_id': 1003L, 'physical_network': None, 'network_type':
'vxlan', 'id': '07d84f9f-831f-4051-a810-1f7211bbeafc'})
```

与 Type Manager 相比，Mechanism Manager 的接口要整齐很多，一种形如“{action}-{object}-precommit”的接口在数据库 session 内调用，另一个形如“{action}-{object}-postcommit”的接口则在数据库提交完成之后调用。

Mechanism Manager 分发操作并具体传递操作到 Mechanism Driver 的方式与 Type Manager

相同，但是一个需要 Mechanism Driver 处理的操作会按照配置的顺序依次调用每一个 Driver 的对应函数来完成的，比如对于需要配置交换机的操作，可能 Open vSwitch 虚拟交换机和外部真实的物理交换机比如 Cisco 交换机都需要进行配置，这个时候就需要 Open vSwitch Mechanism Driver 和 Cisco Mechanism Driver 都被调用进行处理。

在 `neutron/plugins/ml2/managers.py` 文件中，除了 Type Manager 与 Mechanism Manager 之外还定义有 Extension Manager，将 ML2 支持的每个 Extension API 都作为单独的 Extension Driver 来实现。

4. Type Driver

Type Driver 最主要的功能是管理网络 Segment，提供 Provider Segment 和 Tenant Segment（命令行里没有指定任何 Provider 信息时，所创建的就是 Tenant Segment，简单地说，Provider Segment 之外的 Segment 都可以称为 Tenant Segment）的验证、分配、释放等功能。

(1) Flat Type Driver

Flat 网络创建时，必须指定 `PHYSICAL_NETWORK` 信息（比如命令行里指定“`--provider:physical_network`”的值），也就是必须指定物理网络的名字，而且这个名字还必须符合配置文件的要求。而且对于 Flat 网络来说，没有所谓的 `SEGMENTATION_ID`，这是因为 VLAN ID、Tunnel ID 等对于 Flat 网络没有任何意义。Flat Type Driver 会根据上述的要求对 Segment 进行验证。

对于 Flat 类型的网络来说，Segment 的分配很简单，就是将 Type Manager 传递过来的 Segment（从 network 创建的命令里提取出来的结构）保存在数据库里。这个过程会检查数据库里是否已经存在相同的条目，如果有，就说明该 Segment 已经被使用了，这个 Flat 网络的创建就会失败。如果数据库里并没有存在相同的条目，则还需检查配置文件的设置，通常我们需要将需要创建的 Flat 网络的物理网络名称写入配置文件，如果这个名称使用“*”通配符代替，则表示任意的物理网络名称都满足要求。

(2) Tunnel Type Driver

VxLAN 和 GRE 都是 Tunnel 类型的虚拟网络，针对 Tunnel 网络，ML2 又引入了类 `neutron.plugins.ml2.drivers.type_tunnel.TunnelTypeDriver`，这个类继承自 `neutron.plugins.ml2.driver_api.TypeDriver`，除了实现了 Type Driver 要求的接口，还针对 Tunnel 类型网络定义了一些新的接口供 VxLAN 与 GRE Driver 去实现。此外，`TunnelTypeDriver` 类还定义了与 Agent 进行交互的接口：ML2 Plugin 到 Agent 的 `tunnel_update()` 和 Agent 到 ML2 Plugin 的 `tunnel_sync()`。

(3) VLAN Type Driver

VLAN 的管理则与 GRE 与 VxLAN 不同。VLAN 必须指定有 `PHYSICAL_NETWORK`，这是因为 VLAN 必须在主机的某个网络接口（比如 `eth0`）上配置，而 VxLAN 和 GRE 不需要和主机网络接口绑定。

每个物理网络上都可以有 4095 个可用 VLAN ID，即最多可以有 4095 个 Segment。

5. Mechanism Driver

部分 Mechanism Driver, 包括 Open vSwitch、LinuxBridge、Hyperv 等都采用了已有的 Agent, 也就是 ML2 引入前那些 Plugin 所对应的 Agent 来完成具体的操作。这几种 Mechanism Driver 的实现都不是直接从 `neutron.plugins.ml2.driver_api.MechanismDriver` 类继承而来的, 而是又引入了一个新的类 `neutron.plugins.ml2.drivers.mech_agent.AgentMechanismDriverBase` 来针对这种情况。

这种类型的 Mechanism Driver 并没有实现任何 MechanismDriver 类中定义的接口, 它们工作的内容主要都是做 Port Binding 相关的处理。至于与 Agent 之间的交互, 则是由 Ml2Plugin 类之中的两个函数负责: `delete_network()` 与 `update_port()`。

其中, `delete_network()` 完成的事情很单一, 就是将一个 network 删除。其余众多牵涉设备的操作都由 `update_port()` 来通知 Agent 完成。

7.4 Port Binding 扩展

Extension API 有两种方式扩展现有的资源: 一种是为 `network/port/subnet` 增加属性, 比如这里将要介绍的 Port Binding 扩展。另外一种就是增加一些新的资源, 比如 VPNaaS 等。

Extension API 的定义都位于 `neutron/extensions` 目录, 它们的基类以及一些公用的代码则位于 `neutron/api/extensions.py` 文件, 其中类 `ExtensionDescriptor` 是所有 Extension API 的基类, 添加新的资源时需要实现 `get_resources()` 方法, 而扩展现有的资源时, 则只需实现 `get_extended_resources()` 方法。

因为 Port Binding 只是对 port 的扩展并不引入新的资源, 它只需要实现 `get_extended_resources()` 方法即可。

```
# neutron/extensions/portbindings.py

EXTENDED_ATTRIBUTES_2_0 = {
    'ports': {
        VIF_TYPE: { # 未 bind 前, 这个值为 unbond
                    # 如果 host_id 设置后未能 bind 则设置为 binding_failed
                    # 绑定成功之后是具体 VIF 类型
                },
        VIF_DETAILS: { # 提供更详细的信息和功能, 如 port_filter 相关的
                        # security group 或者 anti-MAC/IP spoofing.
                    },
        VNIC_TYPE: { # normal: 虚拟网卡
                     # direct: 直接 passthrough 给 VM 的 SRIOV 网卡
                     # macvtap: Neutron SRIOV 的一种实现
                },
        HOST_ID: { # 端口绑定的 host 主机 id, 由 Nova 设置
```

```

    },
    PROFILE: { # 字典域, 包含诸如"port_filter: True"等类型的信息
    },
}

}

class Portbindings(extensions.ExtensionDescriptor):
    def get_extended_resources(self, version):
        if version == "2.0":
            return EXTENDED_ATTRIBUTES_2_0
        else:
            return {}

```

我们可以看到上面的实现主要就是定义了需要对 port 进行扩展的一些属性, 这些属性使用 `binding:host_id` 的形式指定。

所谓的 Port Binding 可以简单理解为对 port 的信息进行填充扩展。port 在创建时, 只具有 MAC 地址等少量的信息, Nova 会在虚拟机创建时, 为虚拟机指定一个 port, 并通过 nova-scheduler 调度选择一个计算节点用于虚拟机的创建, 此时, 这个 port 也相当于跟随这个新建的虚拟机一起来到了这个计算节点, 于是 port 就拥有了一个属性 `host_id` 表示它所位于的计算节点。

Neutron 通过 Port Binding Extension API 对 port 的信息进行扩展, 比如设置 VIF (Virtual Network Interface), Nova 需要根据 Neutron 设置的 `VIF_TYPE` 通过一些底层的库, 比如 `libvirt` 来创建 VIF 和相关的 Bridge。

在 `binding:host_id` 未被 Nova 有效设置前, `binding:vif_type` 的值应该是 “unbound”。如果 `binding:host_id` 已经有合法值, 但是又不能创建 binding, 那么 `binding:vif_type` 的值应该为 “binding_failed”。

在 ML2 Core Plugin 实现中, 很多代码, 特别是基于已有 Agent 的几个 Mechanism Driver, 大都是在处理 Port Binding 扩展, 这里针对 Port Binding 扩展的介绍也主要是基于 ML2。

针对一个 port 进行 bind 操作时, Mechanism Manager 会依次调用所有的 Driver 来尝试 bind (设置 Port Binding 扩展的那些属性), 对于 Open vSwitch、Linux Bridge 和 Hyperv 这 3 种 Driver 来说, 它们会使用 Nova 设置的 `binding:host_id` 查询保存 Agent 信息的数据库, 来确定节点上是否有对应的 Agent 在运行, 如果有, 则进一步检查 Agent 的配置, 看是否支持所请求 Segment 的 `NETWORK_TYPE`。如果一个 Driver 尝试 bind 成功, 则不再调用后续的 Driver。

上述介绍的只是 Port Binding 的一般情况, Neutron 从 K 版本开始在 ML2 框架下引入了层级式 (Hierarchical) Port Binding, 使实现变得稍微复杂起来。Hierarchical Port Binding 主要想解决的问题如图 7-10 所示。

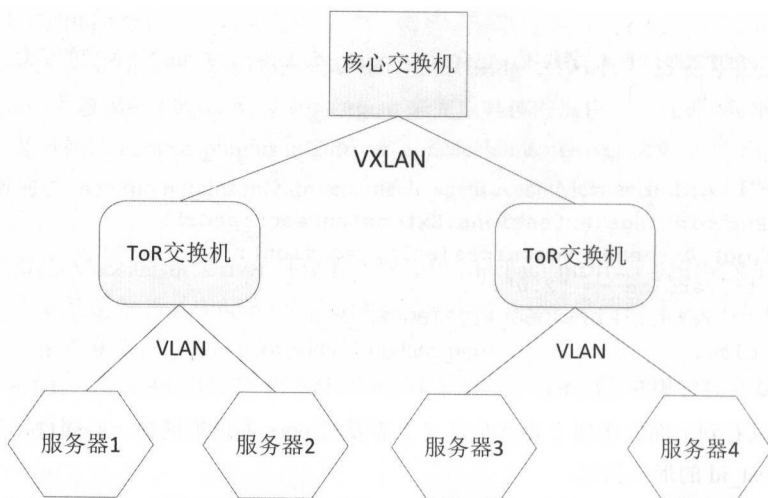


图 7-10 Hierarchical Port Binding

在前面介绍 multi-segment network 时提到一个 provider network 可以由多个物理的 segment 组成。在 Hierarchical Port Binding 的应用场景下，这些 segments 是有层级关系的。比如说一个虚拟网络在顶层上是由 VXLAN 标识的，并且这个 VXLAN 的 segment ID 是静态固定的，但是在 ToR 交换机到服务器的层面上，这些 segments 是由 VLAN 来标识的，并且这些 VLAN ID 可以是 ToR 动态分配的。ToR 负责 VLAN 之间以及 VLAN 和 VXLAN 之间的转发。这样做的好处在于位于一个 ToR 下面的虚拟网络可以通过 VLAN 来承载，效率比较高。如果 ToR 支持将虚拟网络 A 在服务器 1 和服务器 2 之间通过不同的 VLAN ID 承载，那么理论上一个服务器上所能创建的虚拟网络的数量是 4095。如果 ToR 只支持服务器 1 到服务器 2 上的同一虚拟网络用同一 VLAN ID 承载的话，那么连接在同一个 ToR 上的服务器所能共同创建的虚拟网络的最大理论值是 4095。

在上述的这种层级式的虚拟网络拓扑结构下，做 Port Binding 的时候，一个 port 首先要被 ToR 的 ML2 driver 做 partially binding，然后 ToR 的 ML2 driver 会调用 PortContext.continue_binding() 由下一级的 ML2 driver 负责继续做 Port Binding。一般来说，Hierarchical Port Binding 需要网络设备厂商的特殊支持。

对 ML2 来说，Port Binding 有关的操作会涉及它的框架所包含的各个层次：ML2Plugin 类、Mechanism Manager 以及 Mechanism Driver。

1. ML2Plugin 类中的处理

Plugin 主要把握进行 Port Binding 的时机：port 创建时，即 create_port() 执行时，如果已经提供有足够的信息，则需要第一时间进行 bind；由于 update_port() 导致端口信息变化时也需要重新 bind 端口；Agent 通过 RPC 从 Plugin 获取 port 信息的时候，也会尝试 bind 端口，这样

会避免在获取信息的过程中，由于有其他进程更新了 port 但是并没有 bind 而导致 Agent 不能获取到最新的 bind 信息。

```
@utils.transaction_guard
@db_api.retry_if_session_inactive()
def create_port(self, context, port):
    # 数据库操作以及对 port binding 的一些 clean up
    result, mech_context = self._create_port_db(context, port)
    # notify any plugin that is interested in port create events
    kwargs = {'context': context, 'port': result}
    registry.notify(resources.PORT, events.AFTER_CREATE,
                    self, **kwargs)

    try:
        self.mechanism_manager.create_port_postcommit(mech_context)
    except ml2_exc.MechanismDriverError:
        with excutils.save_and_reraise_exception():
            LOG.error(_LE("mechanism_manager.create_port_postcommit "
                          "failed, deleting port '%s'"), result['id'])
            self.delete_port(context, result['id'],
                            l3_port_check=False)

    # REVISIT(rkukura): Is there any point in calling this before
    # a binding has been successfully established?
    self.notify_security_groups_member_updated(context, result)

    try:
        # 尝试做 port binding
        bound_context = self._bind_port_if_needed(mech_context)
    except os_db_exception.DBDeadlock:
        # bind port can deadlock in normal operation so we just cleanup
        # the port and let the API retry
        with excutils.save_and_reraise_exception():
            LOG.debug("_bind_port_if_needed deadlock, deleting port %s",
                    result['id'])
            self.delete_port(context, result['id'])
    except ml2_exc.MechanismDriverError:
        with excutils.save_and_reraise_exception():
            LOG.error(_LE("_bind_port_if_needed "
                          "failed, deleting port '%s'"), result['id'])
            self.delete_port(context, result['id'],
                            l3_port_check=False)

    return bound_context.current
```

```

def _create_port_db(self, context, port):
    attrs = port[attributes.PORT]
    if not attrs.get('status'):
        attrs['status'] = const.PORT_STATUS_DOWN

    session = context.session
    with session.begin(subtransactions=True):
        dhcp_opts = attrs.get(edo_ext.EXTRADHCPOPTS, [])
        port_db = self.create_port_db(context, port)
        result = self._make_port_dict(port_db,
                                       process_extensions=False)
        self.extension_manager.process_create_port(context,
                                                    attrs, result)
        self._portsec_ext_port_create_processing(context,
                                                  result, port)

        # sgids must be got after portsec checked with security group
        sgids = self._get_security_groups_on_port(context, port)
        self._process_port_create_security_group(context,
                                                  result, sgids)
        network = self.get_network(context, result['network_id'])
        # 在数据库创建 binding 信息, 设置 port_id, 设置 vif-type 为 “unbound”
        binding = db.add_port_binding(session, result['id'])
        mech_context = driver_context.PortContext(self,
                                                  context, result,
                                                  network,
                                                  binding, None)

        # 从传递过来的 RESTful 请求参数中获取 host、vnic_type 等信息,
        # 并查询数据库是否已经有相应的条目存在, 如果没有或者有但是与这里
        # 获取到的信息不一致, 则将 vif_type 设置为 UNBOND。
        self._process_port_binding(mech_context, attrs)

    result[addr_pair.ADDRESS_PAIRS] = (
        self._process_create_allowed_address_pairs(
            context, result,
            attrs.get(addr_pair.ADDRESS_PAIRS)))
    self._process_port_create_extra_dhcp_opts(context, result,
                                              dhcp_opts)
    self.mechanism_manager.create_port_precommit(mech_context)
    self._setup_dhcp_agent_provisioning_component(context,
                                                  result)

    self._apply_dict_extend_functions('ports', result, port_db)

```



```

        return result, mech_context
    @utils.transaction_guard
    @db_api.retry_if_session_inactive()
    def create_port(self, context, port):
        # 数据库操作以及对 port binding 的一些 clean up
        result, mech_context = self._create_port_db(context, port)
        # notify any plugin that is interested in port create events
        kwargs = {'context': context, 'port': result}
        registry.notify(resources.PORT, events.AFTER_CREATE,
                        self, **kwargs)

    try:
        self.mechanism_manager.create_port_postcommit(mech_context)
    except ml2_exc.MechanismDriverError:
        with excutils.save_and_reraise_exception():
            LOG.error(_LE("mechanism_manager.create_port_postcommit "
                          "failed, deleting port '%s'"), result['id'])
            self.delete_port(context, result['id'],
                             l3_port_check=False)

    # REVISIT(rkukura): Is there any point in calling this before
    # a binding has been successfully established?
    self.notify_security_groups_member_updated(context, result)

    try:
        # 尝试做 port binding
        bound_context = self._bind_port_if_needed(mech_context)
    except os_db_exception.DBDeadlock:
        # bind port can deadlock in normal operation so we just cleanup
        # the port and let the API retry
        with excutils.save_and_reraise_exception():
            LOG.debug("_bind_port_if_needed deadlock, deleting port %s",
                      result['id'])
            self.delete_port(context, result['id'])
    except ml2_exc.MechanismDriverError:
        with excutils.save_and_reraise_exception():
            LOG.error(_LE("_bind_port_if_needed "
                          "failed, deleting port '%s'"), result['id'])
            self.delete_port(context, result['id'],
                             l3_port_check=False)

    return bound_context.current

def _create_port_db(self, context, port):

```



```

attrs = port[attributes.PORT]
if not attrs.get('status'):
    attrs['status'] = const.PORT_STATUS_DOWN

session = context.session
with session.begin(subtransactions=True):
    dhcp_opts = attrs.get(edo_ext.EXTRADHCPOPTS, [])
    port_db = self.create_port_db(context, port)
    result = self._make_port_dict(port_db,
                                   process_extensions=False)
    self.extension_manager.process_create_port(context,
                                                attrs, result)
    self._portsec_ext_port_create_processing(context,
                                              result, port)

    # sgids must be got after portsec checked with security group
    sgids = self._get_security_groups_on_port(context, port)
    self._process_port_create_security_group(context,
                                              result, sgids)
    network = self.get_network(context, result['network_id'])
    # 在数据库创建 binding 信息, 设置 port_id, 设置 vif-type 为 "unbound"
    binding = db.add_port_binding(session, result['id'])
    mech_context = driver_context.PortContext(self,
                                              context, result,
                                              network,
                                              binding, None)

    # 从传递过来的 RESTful 请求参数中获取 host、vnic_type 等信息
    # 并查询数据库是否已经有相应的条目存在, 如果没有或者有但是与这里
    # 获取到的信息不一致, 则将 vif_type 设置为 UNBOND
    self._process_port_binding(mech_context, attrs)

    result[addr_pair.ADDRESS_PAIRS] = (
        self._process_create_allowed_address_pairs(
            context, result,
            attrs.get(addr_pair.ADDRESS_PAIRS)))
    self._process_port_create_extra_dhcp_opts(context, result,
                                              dhcp_opts)
    self.mechanism_manager.create_port_precommit(mech_context)
    self._setup_dhcp_agent_provisioning_component(context,
                                                  result)

self._apply_dict_extend_functions('ports', result, port_db)
return result, mech_context

```

至于 `update_port()`, Port Binding 的处理逻辑与 `create_port()` 相同, 它们进行端口 bind 时数

数据库 session 已经结束。这么做的原因是,有些 Driver 可能需要在 bind 过程中进行 RPC 通信,此时应避免持有数据库 transaction,但是如此一来就可能会导致 bind 端口时引起竞争。

上述代码中的 `_bind_port_if_needed()` 的目的就是处理这种竞争,它的基本的思路是如果将 binding 信息进行数据库提交的时候,另一个进程已经提交了 binding 信息则使用另一个进程的结果,如果另一个进程仅仅是改变了 binding 信息但是还没有提交,则尝试将新的 binding 信息提交到数据库。

2. Mechanism Manager 的处理

Mechanism Manager 的处理逻辑很简单,仅仅是实现了 `bind_port()` 方法去依次尝试调用每一个注册的 Mechanism Driver,直到成功 bind。但是引入 Hierarchical Port Binding 之后逻辑变得稍微复杂了一些。

```
def bind_port(self, context):
    binding = context._binding
    LOG.debug("Attempting to bind port %(port)s on host %(host)s "
              "for vnic_type %(vnic_type)s with profile %(profile)s",
              {'port': context.current['id'],
               'host': context.host,
               'vnic_type': binding.vnic_type,
               'profile': binding.profile})
    context._clear_binding_levels()
    # 此处的第二个参数“0”表示从 level 0 开始进行 port binding
    if not self._bind_port_level(context, 0,
                                context.network.network_segments):
        binding.vif_type = portbindings.VIF_TYPE_BINDING_FAILED
        LOG.error(_LE("Failed to bind port %(port)s on host %(host)s "
                      "for vnic_type %(vnic_type)s using segments "
                      "%(segments)s"),
                  {'port': context.current['id'],
                   'host': context.host,
                   'vnic_type': binding.vnic_type,
                   'segments': context.network.network_segments})

    # 这是个嵌套函数,从给定的 level 开始一直向下做 binding,直到所有的 segments
    # 全部 bound,或者出错返回
    def _bind_port_level(self, context, level, segments_to_bind):
        binding = context._binding
        port_id = context.current['id']
        LOG.debug("Attempting to bind port %(port)s on host %(host)s "
                  "at level %(level)s using segments %(segments)s",
                  {'port': port_id,
                   'host': context.host,
```

```

        'level': level,
        'segments': segments_to_bind))

if level == MAX_BINDING_LEVELS:
    LOG.error(_LE("Exceeded maximum binding levels attempting to
bind "

        "port %(port)s on host %(host)s"),
        {'port': context.current['id'],
        'host': context.host})

    return False

# 轮询各个 mechanism driver
for driver in self.ordered_mech_drivers:
    # 防止 binding loop 的发生, 即同一个 driver 不能在同一个 segment
    # 的不同 level 上进行 binding
    if not self._check_driver_to_bind(driver, segments_to_bind,
                                      context._binding_levels):

        continue
    try:
        context._prepare_to_bind(segments_to_bind)
        # 调用 driver 的 bind_port() 接口
        # 如果 driver bind 成功的话, 这个 driver 会将 PortContext 的
        # new_bound_segment 设置成刚刚被 bound 的 segment。
        # 并且如果这个 driver 认为还有下一级需要做 port binding 的话
        # 将 next_segments_to_bind 设置成这个 driver 动态分配的
        # segment ID
        # 对应到图 7-10 的例子来说, 当 ToR 的 driver 昨晚 binding 的时候
        # 它会根据自己的配置将为该 port 所在的虚拟网络的 VLAN ID 设置到
        # next_segments_to_bind。然后由下一级 driver 负责在服务器上
        # 将这个 port bind 到这个 VLAN 上面去
        driver.obj.bind_port(context)
        segment = context._new_bound_segment
        if segment:
            # 将目前的 binding 情况写入数据库
            context._push_binding_level(
                models.PortBindingLevel(port_id=port_id,
                                       host=context.host,
                                       level=level,
                                       driver=driver.name,
                                       segment_id=segment))
            next_segments = context._next_segments_to_bind
            # 如果设置了 next_segments_to_bind, 继续在下一个
            # level 上做 binding
            if next_segments:

```

```

        # Continue binding another level.
        if self._bind_port_level(context, level + 1,
                                next_segments):
            return True
        else:
            LOG.warning(_LW("Failed to bind port %(port)s on
"
                            "
                            "host %(host)s at level %(lvl)s"),
                        {'port': context.current['id'],
                         'host': context.host,
                         'lvl': level + 1})
            context._pop_binding_level()
    else:
        # Binding complete.
        LOG.debug("Bound port: %(port)s, "
                  "host: %(host)s, "
                  "vif_type: %(vif_type)s, "
                  "vif_details: %(vif_details)s, "
                  "binding_levels: %(binding_levels)s",
                  {'port': port_id,
                   'host': context.host,
                   'vif_type': binding.vif_type,
                   'vif_details': binding.vif_details,
                   'binding_levels': context.binding_levels})
        return True
except Exception:
    LOG.exception(_LE("Mechanism driver %s failed in "
                      "bind_port"),
                  driver.name)

```

3. Mechanism Driver 的处理

以 Linux Bridge Driver 为例，neutron.plugins.ml2.drivers.mech_agent.AgentMechanismDriverBase 类为使用已有 Agent 的那些 Driver 实现了公用的 bind_port()方法。它的基本执行逻辑是：寻找 port 所在 host 的所有该 Driver 的 Agent(对于 Linux Bridge Driver 来说仅有一个)，并确保 Agent 处于 live 状态，然后通过具体的 Driver 对每个 Segment 进行逐一检查，如果此 Segment 能够提供符合条件 port，则返回 binding 信息给 Ml2Plugin 类。

7.5 Open vSwitch Agent

ML2 Plugin 的主要工作是管理虚拟网络资源，保证数据正确无误，具体物理设备的设置则由 Agent 完成。本节以 OVS Agent (Open vSwitch Agent) 为例进行介绍。

基于 Plugin 提供的信息, OVS Agent 负责在计算节点或者网络节点上, 通过对 OVS 虚拟交换机的管理将一个 network 映射到物理网络。这需要 OVS Agent 去执行一些 Linux 网络和 OVS 相关的配置与操作, Neutron 通过如下两个库提供了最为基础的操作接口, 从而可以通过 Linux Shell 命令完成 OVS 的配置:

```
agent/linux/ovs_lib.py
```

通过 shell 执行各种 ovs-vsctl 操作。

```
agent/linux/ip_lib.py
```

操作 Linux 的 veth, route, namespace 等。

对 ML2 Plugin 来说, OVS 只是 VLAN/GRE/VxLAN 等不同网络拓扑类型的一种底层实现机制。对于 VLAN 类型的网络, 首先面对的问题是属于不同 network 的外部流量进入一个节点时, 如何对其进行隔离。OVS 的 VLAN 功能可以很好地解决这个问题, 但是需要在节点的入口处创建一个 OVS 的 Bridge, 通常这个 Bridge 都会命名为 br-ethx, 同时将物理网络接口 (比如 eth0、eth1 等) 挂接到这个 Bridge 上。

另一个 VLAN 类型网络需要解决的问题是节点内部不同虚拟网络的隔离, 这通过 Local VLAN 来完成, 每个节点内部都可以看做是一个小型的虚拟网络拓扑, 不同的 VM 通过 Linux Bridge 进行桥接, 这些 Linux Bridge 又会挂接在一个内部的 OVS Bridge (通常命名为 br-int) 上, 基于这个 OVS Bridge 通过内部的 VLAN 将属于不同 network 的 VM 进行二层流量隔离, 对应的 VLAN ID 又称为 LVID(Local VLAN ID)。此外, 对于网络节点来说, 除了 Local VLAN, 还需要利用 Linux Network Namespace 进行网络协议栈的隔离。

我们需要一些配置信息来帮助 Neutron 建立具体的虚拟网络, 典型的 OVS 配置如下:

```
[ml2]
tenant_network_types = vlan
mechanism_drivers = openvswitch

[ml2_type_vlan]
network_vlan_ranges = physnet1:300:500

[ovs]
bridge_mappings = physnet1:br-eth
```

基于这个配置, 节点内虚拟网络的拓扑如图 7-11 所示。

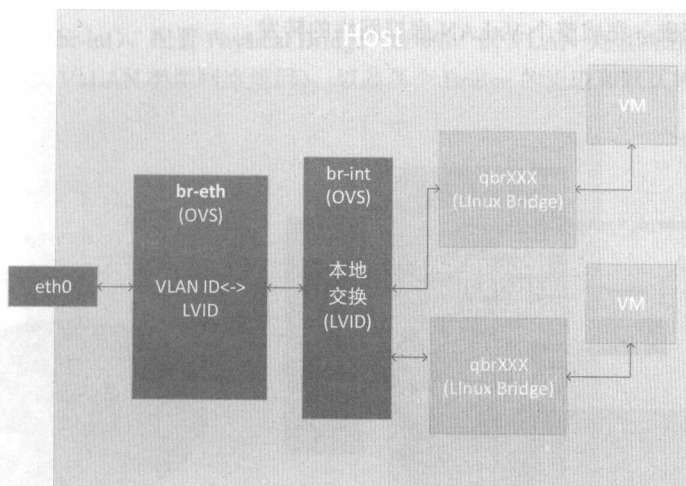


图 7-11 VLAN 类型网络节点内虚拟网络拓扑

br-eth 完成外部物理 VLAN ID 到内部 LVID 的映射，也就是说，会将外部过来的数据包中的 VLAN ID 替换为内部的 LVID。br-int 则负责处理 Local VLAN 的二层交换。qbrXXX 为 Linux Bridge，这个 Bridge 以及 VM 与其的连接都由 Nova 进行设置，并不由 Neutron 负责。

而对于 VxLAN 类型的网络来说，同样也需要解决节点内部不同虚拟网络之间的隔离问题，解决这个问题的方式也与 VLAN 类型网络相同。此外，VxLAN 类型网络主要解决的问题是 OVS 网桥无法从隧道学习 MAC 地址，因此仍然需要为每一个加入 VxLAN 网络的节点建立一个 Tunnel Port，需要和 GRE 一样的学习机制来获取远端 MAC 和 Tunnel Port 的对应关系。

只是因为 OVS 网桥无法从隧道学习 MAC 地址，所以才采用了和 GRE 相同的学习策略。当报文从 br-int 进入 br-tun (VxLAN 同样在节点入口处创建一个 Bridge 来负责外部流量的隔离) 后，通过目的 MAC 应该找到一个单播 IP 和 VNI (VxLAN ID)，OVS 不能自动学习这个对应关系，需要为每个 VxLAN 端点 (VxLAN 隧道两端的节点) 建立一个 Tunnel Port，然后通过 OVS 流表规则学习到这个 Tunnel Port。

当单播报文从远端进入 br-tun 的时候，就可以确信，这个报文的源 MAC 地址对应的 VM 肯定位于此 Tunnel Port 所连接的远端主机，这就是发送报文需要的 OVS 端口。将这个对应关系作为一个规则写入 OVS 的一个流表，即可为反向的流量提供 MAC 到 Tunnel Port 的映射 (即远端 IP 和 VNI)。

这个过程和一般交换机 MAC 地址学习的过程极为类似，学习过程中使用的 OVS 流表是 LEARN_FROM_TUN，学习到的规则存储于 UCAST_TO_TUN。

单播发送问题可以通过上述的学习过程建立映射，那么针对未知单播或广播报文又该如何处理？答案是通过多播或者单播 (依据配置) 发送到所有的 Tunnel Port 上。

图 7-12 所示为 VxLAN 类型网络节点内虚拟网络的拓扑。在 Tunnel Port 学习的基础之上

辅以少量其他流表，完成整个 VxLAN 虚拟网络的转发。

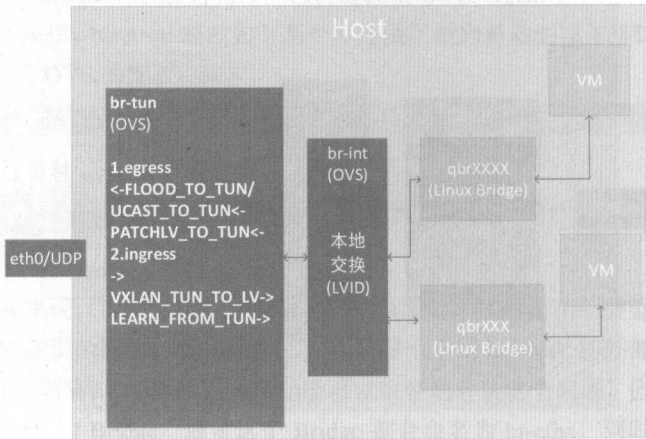


图 7-12 VxLAN 类型网络节点内虚拟网络拓扑

- egress 方向（从节点内部到物理网络）上，从 br-int 进入 br-tun 的数据报文首先由流表 PATCH_LV_TO_TUN 根据报文是否为单播，定向到 UCAST_TO_TUN 或者 FLOOD_TO_TUN。UCAST_TO_TUN 内是上述学习到的 MAC 与 Tunnel Port 的映射规则，而 FLOOD_TO_TUN 则是将报文发送到所有属于虚拟子网的 Tunnel Port。
- ingress 方向（从 Tunnel Port 到节点内部网络），从 Tunnel Port 进入的报文经过流表 VXLAN_TUN_TO_LV，此流表负责查找 VNI 对应的 LVID 填入报文，之后将报文送到表 LEARN_FROM_TUN 进行 MAC 地址学习。

与图 7-11 所示不同，物理网络接口 eth0 与 br-tun 之间并没有连接，对于 VxLAN 来说，eth0 接收到外部进入的网络流量后，Linux 网络协议栈会将其转交给 br-tun 进行处理，因此也并不需要配置 eth0 与 br-tun 之间的 bridge-mapping。

1. Agent 初始化

从 setup.cfg 文件可知，OVS Agent 的入口位于 neutron/plugins/openvswitch/agent/ovs_neutron_agent.py 文件。OVS Agent 在初始化阶段会建立基本完整的虚拟网络环境，建立 VLAN 和 Tunnel 转发所需要的主要流表和默认规则。假定采用如下配置：

```
[m12]
type_drivers = vxlan,vlan,local
tenant_network_types = vxlan
[m12_type_vxlan]
vni_ranges=1:1000
```

在此配置下，Agent 初始化所建立的各种设施如图 7-13 所示，包括建立 RPC 通道、配置

Integration Bridge (br-int)、配置 Physical Bridge (br-eth, 供 VLAN 类型网络使用) 和 Tunnel Bridge (br-tun, 供 VxLAN 类型网络使用), 以及各个 Bridge 的流表和默认规则。

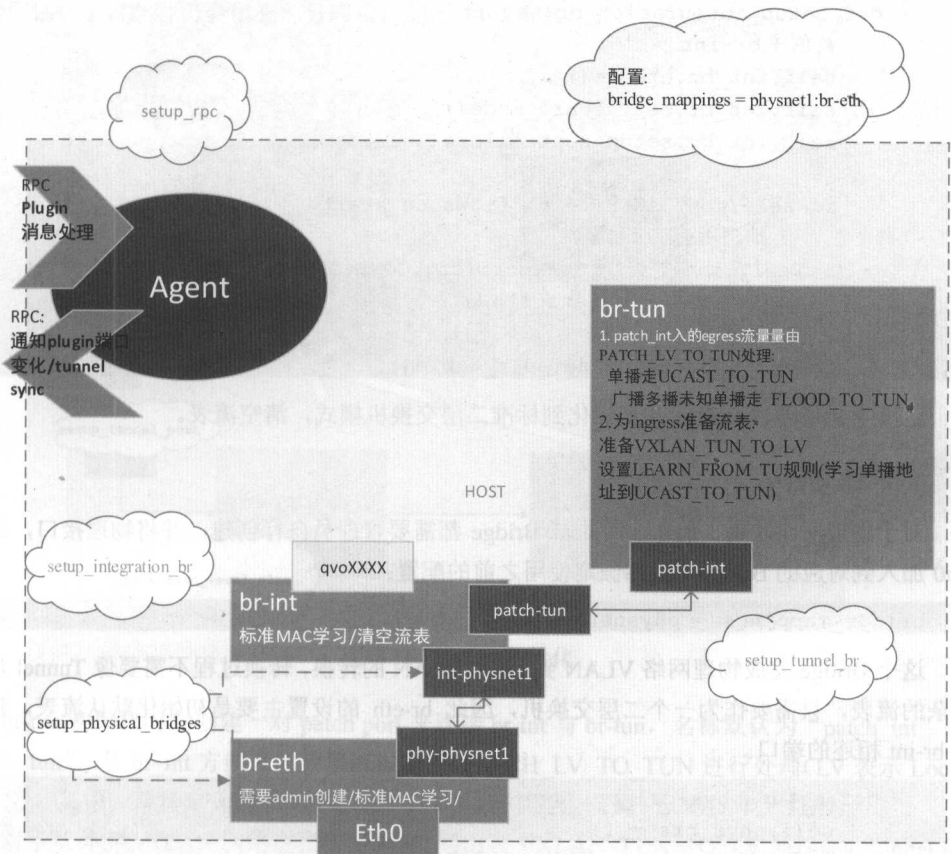


图 7-13 OVS Agent 初始化

2. Agent 与 Plugin 的 RPC 通信

Agent 与 Plugin 直接通过 RPC 通道通信。Plugin 需要通知 Agent 删除 network 以及更新 port, Agent 则向 Plugin 汇报端口 up/down 以及进行 Tunnel sync。

一个交互通道有一个发送方与一个接收方, ML2 Plugin 初始化的时候会建立两个 RPC 通道。其中一个用于通知 Agent 端口变化或者 network 被删除, 另一个用于接收 Agent 请求消息, 为 Agent 提供各种所需信息, 以及接收端口的 up/down 事件。Agent 同样需要两个 RPC 端点, 分别与 Plugin 相对应。

3. br-int 创建与初始化

br-int 用于 Local VLAN 的二层交换，功能简单。

```
def setup_integration_br(self):
    # 创建 br-int
    self.int_br.create()
    self.int_br.set_secure_mode()
    self.int_br.setup_controllers(self.conf)

    if self.conf.AGENT.drop_flows_on_start:
        # 清理所有端口和流表
        self.int_br.delete_port(cfg.CONF.OVS.int_peer_patch_port)
        self.int_br.delete_flows()
    # 创建默认表
    self.int_br.setup_default_table()
```

这里主要是创建 Bridge 并初始化到标准二层交换机模式，清空流表。

4. br-eth 初始化

对于物理网络映射，每个 Physical Bridge 都需要管理员自行创建，并将物理接口，比如 eth0 加入到对应的 Bridge，仍然假定使用之前的配置：

```
bridge_mappings = physnet1:br-eth
```

这个 Bridge 完成物理网络 VLAN 到 Local VLAN 的转换，转换过程不需要像 Tunnel 那样复杂的流表，只需要作为一个二层交换机，因此 br-eth 的设置主要是初始化默认流表，建立与 br-int 相连的端口。

```
def setup_physical_bridges(self, bridge_mappings):
    self.phys_brs = {}
    self.int_ofports = {}
    self.phys_ofports = {}
    ip_wrapper = ip_lib.IPWrapper()
    ovs = ovs_lib.BaseOVS()
    ovs_bridges = ovs.get_bridges()
    # 针对 bridge_mappings 内的每个物理网络
    for physical_network, bridge in six.iteritems(bridge_mappings):

        br = self.br_phys_cls(bridge)
        br.create()
        br.set_secure_mode()
        br.setup_controllers(self.conf)
        if cfg.CONF.AGENT.drop_flows_on_start:
            br.delete_flows()
        br.setup_default_table()
```

5. br-tun 初始化

整个 Tunnel 转发所需要的表项大部分在初始化阶段会设置完成，在 Tunnel 建立、拆除以及分配回收 LVID 时仍会填充部分表项，如图 7-14 所示。

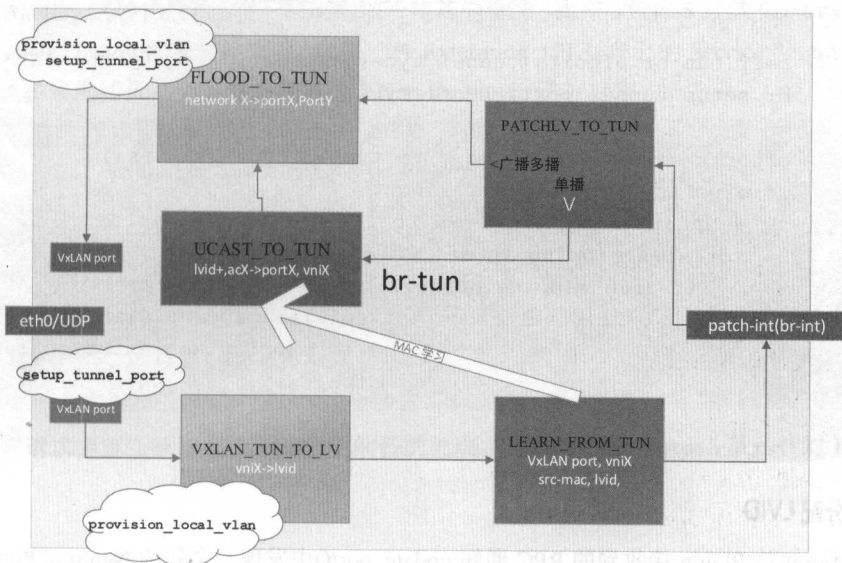


图 7-14 br-tun 初始化

br-tun 初始化时会创建一对 patch port 来连接 br-int 与 br-tun，名称默认为“patch_int”与“patch_tun”，从 br_int 方向来的数据流根据表 PATCH_LV_TO_TUN 进行处理(LV 表示 Local VLAN)，其中，单播定向到表 UCAST_TO_TUN，多播定向到 FLOOD_TO_TUN。

对于从外部 VxLAN 网络进来的数据流，则由表 VXLAN_TUN_TO_LV 处理，将 Tunnel ID 映射为 LVID，表 LEARN_FROM_TUN 用于自动学习远端 MAC，学习结果会填入 UCAST_TO_TUN 流表。

6. 创建 Tunnel Port

当 Plugin 通知 Agent 一个 tunnel update 事件时，Agent 检查本地有无对应的 Tunnel 端口，没有则需要设置对应的 Tunnel 端口，并设置 br-tun 流表规则，使得所有来自此端口的流量都要经过 VXLAN_TUN_TO_LV 处理。

```
def _setup_tunnel_port(self, br, port_name, remote_ip, tunnel_type):
    # port_name 为添加的 Tunnel 端口, tunnel_type 为 VxLAN,
    # vxlan_udp_port 默认为 4789
    ofport = br.add_tunnel_port(port_name,
                                remote_ip,
```

```

        self.local_ip,
        tunnel_type,
        self.vxlan_udp_port,
        self.dont_fragment,
        self.tunnel_csum)

    self.tun_br_ofports[tunnel_type][remote_ip] = ofport
    br.setup_tunnel_port(tunnel_type, ofport)

    ofports = self.tun_br_ofports[tunnel_type].values()
    if ofports and not self.l2_pop:
        # Update flooding flows to include the new tunnel
        for vlan_mapping in self.vlan_manager:
            if vlan_mapping.network_type == tunnel_type:
                br.install_flood_to_tun(vlan_mapping.vlan,
                                         vlan_mapping.segmentation_id,
                                         ofports)

    return ofport

```

7. 分配 LVID

当 Agent 从 Plugin 接收到的 RPC 通知 `update_port()` 中发现一个新的 Neutron Port 时，会为此 VNI 分配一个 LVID。此时，如果 Tunnel 没有建立，则 LVID 没有对应的出端口，将不会建立流表。如果 Tunnel 已经建立，则需要针对 ingress 和 egress 方向为这个新的 LVID 增加转发规则。

8. L2Population

采用 L2Population 后，基于 VxLAN 或者 GRE 组网时可以不依赖于广播流和 Flood 操作来学习 OVS 流表，从而减少广播/组播流量。

首先 ARP Proxy 通过 Linux 现有机制，来应答本地 VM 请求远端 VM MAC 地址的 ARP 报文，从而减少广播流量。ARP Proxy 使能后不再进行 Flood，要有其他手段填充 FDB 表（OVS 流表）。基本工作原理是 Agent 在获取到新的 port 信息后，会将 MAC 地址更新给 Plugin，Plugin 将完整的 FDB 同步到所有同网络的 Agent，包括发布此 port 信息的 Agent。

7.6 Service Plugin

Neutron 中除了 network/port/subnet 这几个核心资源，其他都被作为 Extension API 进行实现。随着 ML2 的成熟和体系架构的演变，Extension API 的实现演变为两种方式，一种是实现在某个 Core plugin 内，比如 ML2 内的 Port Binding、Security Group 等；另一种就是使用 Service

Plugin 的方式。

与之对应的，收到针对 Extension API 的用户请求后，会将其分发给支持这个扩展的 Core Plugin 或对应的 Service Plugin。

Externion API 有两种方式扩展现有的资源，一种是为 netowrk/port/subnet 增加属性，比如 Port Binding 这种扩展；还有一种是增加一些新的资源，比如 VPNaaS 或者 Security Group。

从 Paste Deploy 配置文件来看，如果一种 Externion API 有自己的 Service Plugin 实现，则所有对该扩展的请求都会优先分发到这个 Service Plugin 进行处理：

```
[composite:neutronapi_v2_0]
use = call:neutron.auth:pipeline_factory
noauth = request_id catch_errors extensions neutronapiapp_v2_0
keystone = request_id catch_errors authtoken keystonecontext extensions
neutronapiapp_v2_0

[filter:extensions]
paste.filter_factory =
neutron.api.extensions:plugin_aware_extension_middleware_factory
```

用户请求会首先经过一些 WSGI 中间件的处理，其中的“extensions”就是针对 Externion API。

7.6.1 Firewall

FWaaS 提供虚拟防火墙给租户网路，从 H 版本开始支持基于 Linux IPTables 的 FWaaS，至今仍在发展中，比如目前正在积极推进地针对 McAfee 的支持。

Neutron 已有的网络安全模块是 Security Group，但是其支持的功能有限，且只能对单个 port 有效，不能满足很多需求，比如租户不能选择性地应用 rule 到自己的网络。

FWaaS 定义的数据模型有 3 个（对应数据库中的 3 个表）：firewall、policy 与 rule。租户可以创建 firewall，每个 firewall 可以关联一组 policy，而 policy 是 rule 的有序列表。policy 相当于一个模板，由 admin 创建的 policy 可以在租户之间共享。rule 不能直接应用到 firewall，必须加入 policy 后才能和防火墙关联。

创建 rule:

```
$ neutron firewall-rule-create --protocol {tcp,udp,icmp,any} \
--source-ip-address SOURCE_IP_ADDRESS \
--destination-ip-address DESTINATION_IP_ADDRESS \
--source-port SOURCE_PORT_RANGE --destination-port DEST_PORT_RANGE \
--action {allow,deny,reject}
```

创建 policy:

```
$ neutron firewall-policy-create --firewall-rules
```

```
"FIREWALL_RULE_IDS_OR_NAMES" myfirewallpolicy
```

多个防火墙 rule 的 ID 用空格分隔, 注意 rule 的排列顺序是很重要的。我们能够创建一个不包括任何 rule 的 policy, 并随后为其添加 rule。

创建 firewall:

```
$ neutron firewall-create FIREWALL_POLICY_UUID
```

Firewall Service Plugin 的实现借鉴了 ML2 的结构化思路, 也将整个框架划分为 Plugin、Agent 和 Driver 这 3 个部分。但是与 ML2 不同的是, Firewall Plugin 的 Driver 并不是 Plugin 的组成部分, 而是 Agent 的组成部分。

Firewall Plugin 的 Driver 是给 Agent 用来操作具体的 Firewall 设备的, 因此类似于 Firewall 设备对应的设备 Driver, 比如 Linux IPTables Driver。Firewall Plugin 也没有独立运行的 Agent 进程, Firewall Agent 以 Mixin 模式集成到 L3 Agent 之内在网络节点上运行。Firewall Agent 没有内部的中间状态需要保存或者记录到数据库, 它只起到一个中间人的作用。Firewall Service Plugin 的实现框架如图 7-15 所示。

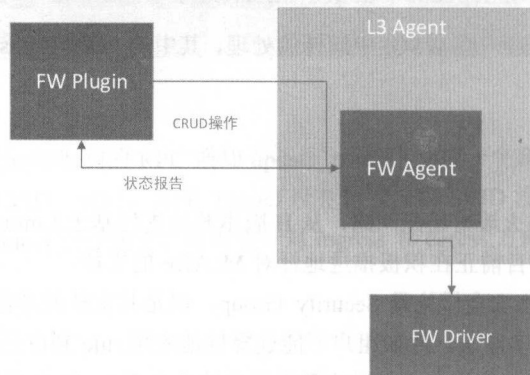


图 7-15 Firewall Service Plugin 实现框架

Firewall Agent 内嵌于 L3 Agent, 响应 Firewall Plugin 的操作, 收集 Driver 所需要的信息, 进而转交给 Firewall Driver 去操作具体的 Firewall 设备。

Firewall 的位置和 Router 类似, 它也必须支持 DVR (Distributed Virtual Router)。在 DVR 情景下, Firewall rule 在计算节点上需要安装到 FIP (floating IP) namespace, 在控制节点上需要安装到 SNAT namespace 内。

7.6.2 LoadBalance

LBaaS 提供在 VM Instance 之间做负载均衡的能力。LBaaS 长期的目标是提供一组 API 让用户可以在不同的 LB 后端实现之间实现无缝切换。当前 LBaaS 是项目 Neutron 的一部分,

不过，已经有计划为其创建单独的 OpenStack 项目。

一个典型的 LB 场景是租户需要把 Web 应用部署到 n 个位于同一个虚拟网络内的 VM 组成 HA，每个 VM 内都会运行 Web Server，比如 Apache，那么 LB 需要提供一个唯一的公用的 IP 来访问 HA 内的这些 VM，并将指向这个 IP 的流量负载均衡分配给各个 VM。更进一步，用户可能需要部署 m 个 HTTP Server 和 n 个 HTTPS Server，甚至这些 Server 并不在同一个虚拟网络内，也需要使用同一个 IP 来访问。

总体上，LBaaS 主要提供以下功能：

- 将网络流量负载均衡到 VM。
- 在不同协议，比如 TCP/HTTP 之间做负载均衡。
- 监控应用和服务的状态。
- 链接限制，入站流量既可以根据链接限制做 shape，也可以作为负载控制，防 DoS 攻击的手段。
- Session persistence，通过源 IP 或者 Cookie 路由，保证将请求送到负载池中的指定虚拟机。

使用 LoadBalance Plugin 配置负载均衡需要完成 3 个任务：首先创建负载均衡，创建一个 LB listener 跟它关联；然后为计算池并添加几个成员；最后创建 health monitor。

创建一个 LB：

```
$ neutron lbaas-loadbalancer-create --name test-lb private-subnet
```

创建一个 LB http listener：

```
$ neutron lbaas-listener-create \  
--name test-lb-http \  
--loadbalancer test-lb \  
--protocol HTTP \  
--protocol-port 80
```

创建一个 LB pool：

```
$ neutron lbaas-pool-create \  
--name test-lb-pool-http \  
--lb-algorithm ROUND_ROBIN \  
--listener test-lb-http \  
--protocol HTTP
```

创建一个 LB pool members：

```
$ neutron lbaas-member-create \  
--subnet private-subnet \  
--address 192.168.1.16 \  
--protocol-port 80 \  
test-lb-pool-http
```

```
$ neutron lbaas-member-create \
--subnet private-subnet \
--address 192.168.1.17 \
--protocol-port 80 \
test-lb-pool-http
```

创建 health monitor:

```
$ neutron lbaas-healthmonitor-create \
--delay 5 \
--max-retries 2 \
--timeout 10 \
--type HTTP \
--pool test-lb-pool-http
```

LoadBalance Service Plugin 同样采用了结构化的实现方式, 将整个框架划分为 Plugin、Plugin Driver、Agent 和设备 Driver。LoadBalance Service Plugin 的实现框架如图 7-16 所示。

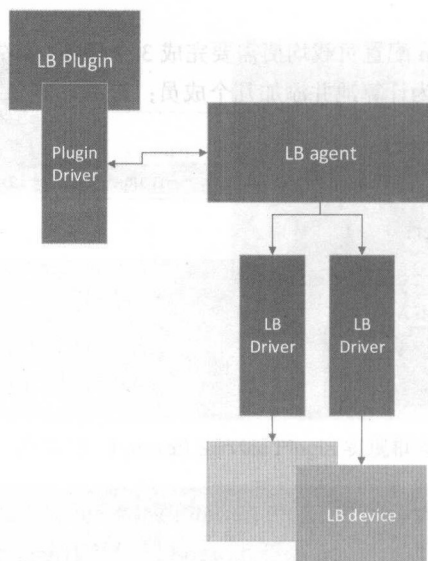


图 7-16 LoadBalance Service Plugin 实现框架

Plugin 负责完成数据存储、请求验证, 以及调度, 其调度功能的目的是为一个 LB pool 分配一个 Agent, 换句话说, 就是分配一个 LB 设备来负责一个 LB pool 的负载均衡。在 Plugin 侧有一个 Plugin Driver, 负责收集信息并将其发送至选定的 Agent。Agent 是独立的服务进程, 管理具体 LB Device。设备 Driver 是将统一的 LBaaS 数据模型部署为特定供应商的模型, 并负责配置 LB 设备。

LoadBalance Plugin 支持用户在多个 LB 设备之间进行自由选择, Plugin 部署在网络节点,

而负责进行负载均衡的 LB 设备可能有多个。这个 LB 设备既可以是 VM，也可以是专用设备。每个设备都需要一个 Agent 进行管理。Plugin 会为 LB pool 选择调度一个 active 的 Agent。

7.7 Neutron 热点话题

鉴于网络系统的特殊性，Neutron 目前作为核心项目之一，仍然问题较多，未来需要拓展的地方同样也有很多，一些热门的话题也在社区里反复进行着讨论，这里仅仅列举有限的几个。

7.7.1 DVR

在一个 Neutron 部署范围内，跨虚拟网络的 VM 之间的流量习惯称为东西流量。虚拟网络的 VM 和经由 DNAT (Destination NAT) 与外部网络的数据交换，习惯称为南北流量。在比较大量的数据流量压力下，集中的 Virtual Route 是一个瓶颈和单点故障隐患。东西数据流量经过网络节点，南北数据流量也经过网络节点，并且相互影响。

DVR (Distributed Virtual Switch) 将增强的 L3 Agent 部署到每一个计算节点上。SNAT (Source NAT) 则仍然部署到网络节点上。这样东西向流量不需要通过网络节点，可以从一个 Hypervisor 直接到达另一个 Hypervisor。

这种架构带来很多优势：

- 东西向流量吞吐量增加。
- 高东西流量下 VM 平均带宽增加。
- 南北向流量和东西向流量不再互相干扰。
- 当东西向流量在同一个 Hypervisor 上，就不会走过不必要的路径。

7.7.2 SDN

SDN 致力于实现一个 Control Plane (控制平面) 和 Data Plane (数据平面) 分离的网络架构，并使之标准化。

最初的原型产生于 2006 年的斯坦福大学，学生 Martin Casado 领导了一个名叫 Ethane 的项目。该项目的目的是通过一个集中式的控制器，管理定义整个网络的安全策略。随后基于这一最初想法 Nick McKeown 教授等人提出了 OpenFlow 的概念，并逐渐发展为完备的 SDN。2011 年成立的开放网络基金会 ONF，致力于推广 SDN 和制定规范。

可以这样简述 SDN，在现有设备中配备 OpenFlow 规范的接口软件，随后一个集中的控制器就可以操作所有这些设备，并通过 API 呈现一个统一接口的网络接口给 App。SDN 典型架构如图 7-17 所示。

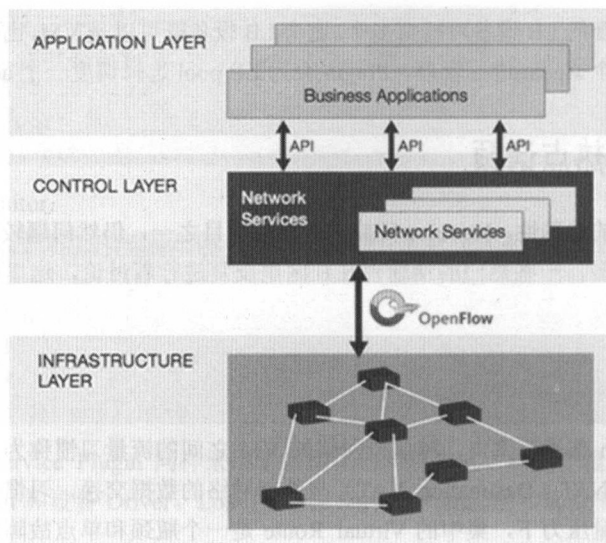


图 7-17 SDN 典型架构

最典型的 SDN APP 就是 NaaS（Networking as a Service），比如 OpenStack 的 Neutron 项目。SDN 控制器呈现给 APP 的接口习惯称为北向接口。而 OpenFlow 承担的从控制器到设备的接口习惯上称为南向接口。

利用 Neutron 灵活可扩展的设计框架，SDN Controller 可以以插件的方式提供支持，已经有部分相关的实现，比如 `neutron/plugins/ibm/agent/sdnve_neutron_agent.py`。

7.7.3 NFV/SRIOV

ETSI（the European Telecommunications Standards Institute）于 2012 年 11 月成立了专门用于讨论 NFV 架构和技术的 ISG（Industry Specification Group，行业规范组），目标是制定一套虚拟化规范，使网络功能不必运行在定制的硬件上，从而可以基于大量工业标准的服务器构建网络。

大量运营商参与到 NFV 规范制定中，成为 NFV 强有力的驱动力量。NFV 模型如图 7-18 所示。

NFV 要求 OpenStack 支持一系列新的特性以便支撑 NFV 定义的一组用户案例。在网络和计算方面，提高 VM 性能，提高网络吞吐率是一个非常重要的指标。

OpenStack 中的 SRIOV，主要解决两个问题：第一个是 SRIOV 网卡如何分配给 VM，第二个是如何配置 SRIOV 网卡使之和 OpenStack 虚拟网络相结合。

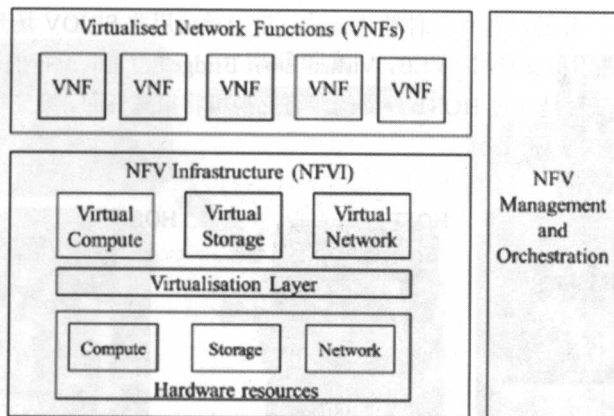


图 7-18 NNF 模型

Openstack SRIOV 虚拟网络囊括了两个领域的虚拟化技术。第一个是系统虚拟化，包括系统提供的 SRIOV VF 支持，Hypervisor/OS 提供的 Macvtap 虚拟网卡等。另一个领域是网络虚拟化技术，包括 VEB、802.1QBG、HW VEB 等。

在一个 SRIOV 已经使能的机器上，通过 lspci 命令可以看到网卡以及通过 SRIOV 技术划分出来的虚拟网络设备。

```
#lspci

04:00.3 Ethernet controller: Intel Corporation I350 Gigabit Network
Connection (rev 01)
04:10.0 Ethernet controller: Intel Corporation I350 Ethernet Controller
Virtual Function (rev 01)
04:10.1 Ethernet controller: Intel Corporation I350 Ethernet Controller
Virtual Function (rev 01)
04:10.2 Ethernet controller: Intel Corporation I350 Ethernet Controller
Virtual Function (rev 01)
```

第一行显示的是原始的支持 SRIOV 的网卡设备，叫做 PF (Physical Function)，虚拟出的设备叫做 VF (Virtual Function)。通过 SRIOV 技术，一个高性能的物理 PCI 设备被划分成多个 VF。在这个例子里每一个 VF 看起来就像一个独立的网卡，但是这些 VF 和 PF 共享同一个以太网端口。

对 HOST 来讲，网络虚拟化要解决虚拟网络在 HOST 内的隔离和互通两个问题。SRIOV 使得每个 VM 有专有硬件资源，不仅使网络层面的隔离更彻底，而且拥有高性能低 CPU 占用率。互通就没有那么直观了，由于 VM 直接操作专属网卡，转发同一个 HOST 内同一个虚拟子网的 VM 之间的报文就成问题。

如图 7-19 所示，互通问题有 3 种解决方法。第一种适用非 SRIOV 场景，同一个虚拟子网内的 VM 通过软件虚拟交换机（VEB，Virtual Ether Bridge）互通，流行的 VEB 主要是 OVS 和 Linux Bridge。第二种是通过 HOST 外界的应将交换机将报文环回。第三种通过内置于网卡内部的硬件 VEB 将报文环回。

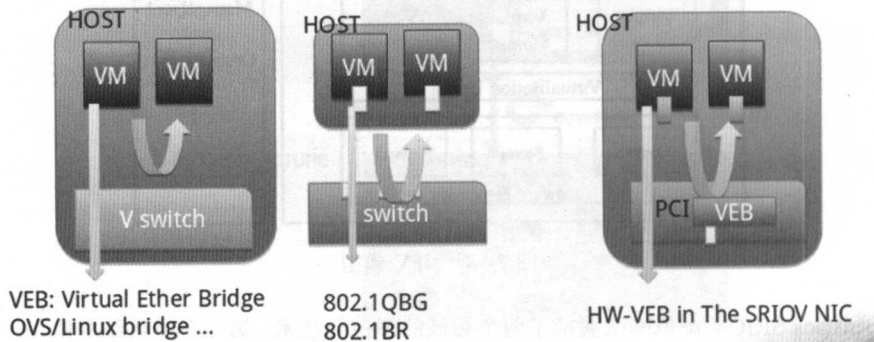


图 7-19 虚拟网络的桥接模式

OpenStack 支持 SRIOV 在虚拟网络内的应用问题，不仅要考虑网络虚拟化技术，还要考虑在计算机系统领域内如何使用 SRIOV 网卡。在系统内，Linux 有两种不同的方式使用 SRIOV 虚拟网卡，第一种叫做 Macvtap，是一种操作系统虚拟化的网卡，但是结合了 SRIOV VF。第二种是直接将 PCI 设备（VF）分配给 VM，就是 Direct 模式。OpenStack 要做的事情比较简单，一个是解决提供给最终用户的界面问题，另一个是配置 Macvtap 或者 Direct 模式。

系统领域的事情是 Nova 负责，而虚拟网络端则是由 Neutron 负责。具体来讲，使用哪种网络虚拟化技术，怎么分配给 VM，都是 Neutron 负责提供信息，而 Nova 负责给 VM 分配和设置 SRIOV VF。

Nova 需要分配 VF，首先要发现系统中可用的 PCI 信息，包括 VF，这样才能随后给 VM 调度，分配合适的 VF，这个 PCI 发现的过程由 Nova 完成。如图 7-20 所示，Nova 启动 VM 时，会根据 VM 相关联的网络或者 Neutron 端口信息，向 Neutron 查询端口类型。如果是 Macvtap 或者 Direct，就意味着需要给 VM 分配一个 VF。分配这个 VF 有一个天然的限制条件，就是这个 VF 对应的 PF 必须接入到 Neutron 虚拟网络对应的物理网络中去，这需要 Neutron 和 Nova 的配置互相协调。一旦拿到这些信息，Nova 在调度 VM 之前就拥有了分配和配置 VF 接口的所有信息，Nova 调度器可以根据这些信息寻找合适的 host，并在启动 VM 时通知 Hypervisor 配置 VF 给这个 VM。

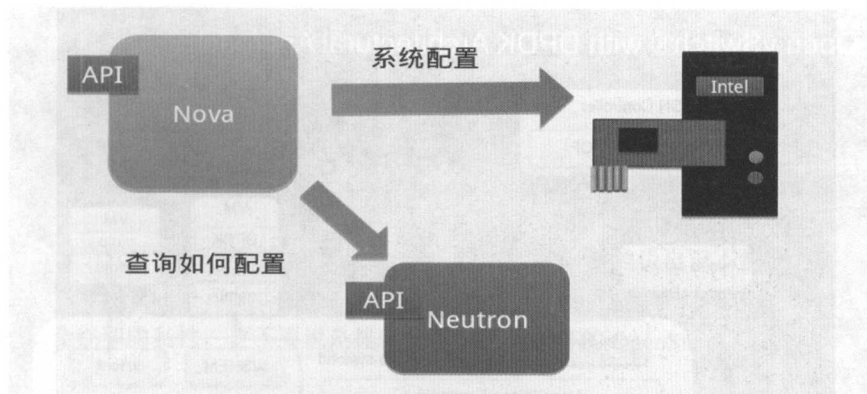


图 7-20 Nova 和 Neutron 的配合

当 EVB 技术与 SRIOV 一起与 Neutron 结合时，有两个问题要解决。首先是这个 VF 如何分配给 VM，其次为是否支持某种 VEB 协议。

为此 SRIOV 扩展了 Port Binding 的属性，增加“binding:vnic_type”，支持“direct”与“macvtap”两种方式将 VF 分配给 VM。“direct”模式下，VF 直接 passthrough 给 VM 使用。而“macvtap”模式下 VF 为 macvtap driver 控制，VM 获得一个该 VF 对应的虚拟网络接口。macvtap 模式支持热迁移和 Security Group。

为支持可以选择 VEB 协议，增加了“binding:vif_type”属性，可以选择支持 802.1QBG、802.1QBH 与 HW_VEB 等。

7.7.4 OVS 和 DPDK

DPDK (Data Plane Development Kit) 提供高性能包处理库和用户空间驱动程序。DPDK 不提供网络协议栈，不提供网络功能，比如三层交换、IPSec、firewalling 等，可以参考示例开发这些网络功能。更详细的文档可以参见 <http://dpdk.org/>。

自 Open vSwitch(OVS)2.4 版起，我们将可在 OVS 中使用 DPDK 优化的 vHost 路径。OVS 自 2.2 版开始提供 DPDK 支持，将 DPDK 与 OVS 结合使用可为我们带来诸多性能优势。与其他基于 DPDK 的应用相同，我们可以在 OVS 中看到网络包吞吐量显著提升，延迟显著降低。图 7-21 所示为 Open vSwitch 与 DPDK 结合之后的架构。

此外，DPDK 包处理库还对 OVS 内的多个性能热点区域进行了优化。比如，转发平面进行了优化，能够作为单独的 vSwitch 后台程序在用户空间内运行（虚拟交换）。实施 DPDK 优化的 vHost 客户机界面，虚拟机—虚拟机或物理机—虚拟机—物理机类型使用案例可获得出色的性能。

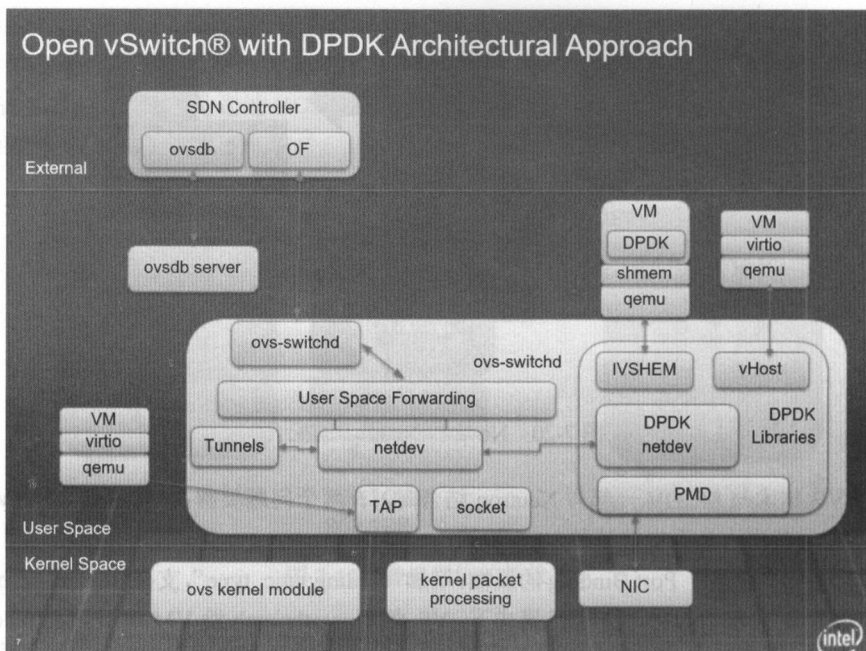


图 7-21 Open vSwitch with DPDK Architectural Approach

Neutron 在 OVS ML2 driver 使用 DPDK vhost-user 接口支持 OVS + DPDK。OVS Agent 检测 Open vSwitch 是否支持 DPDK 并且把这些信息传送到 ML2 driver，ML2 driver 使用这个信息来选择正确的虚拟接口类型和绑定细节。配置如下：

```
[OVS]
datapath_type=netdev
vhostuser_socket_dir=/var/run/openvswitch
```

当 OVS 运行时支持 DPDK 并且 `datapath_type` 设置成 `netdev`，OVS ML2 driver 就使用 `vhost-user` VIF 类型并且把必要的绑定信息来使用 OVS+DPDK 和 `vhost-user` 套接字。这些包括 `vhostuser_socket_dir` 设置。

安全是每个软件无法回避的问题。没有哪款软件可以轻易地说自己不需要考虑安全性因素，当然也没有任何软件产品可以解决所有的安全性问题。即便一个小成本软件也要考虑终端用户的安全性和隐私性，更不要说是提供云基础架构服务的 OpenStack。

8.1 OpenStack 安全概述

随着云计算的愈发火热，参与的人越来越多，政府部门也好，大小企业也罢都跟随着参与到这股浪潮中来。有投资当然是希望能得到回报，而用户作为消费者选择云平台，当然也是因为云计算相对于传统方式能给自己带来费用上的节省，能获得快捷服务的体验。越来越多的用户使用各种云产品就意味着越来越多的数据会存储于“云”端，安全性该如何得到保障无疑会成为摆在云提供商以及终端用户面前的一个非常现实的问题。

2014 年年初携程信息泄露与 2014 年下半年 iCloud 的照片泄露，像一泼冷水浇在云计算这把火上。如何让更多的终端用户放心地参与到云计算中来，依然任重道远。那么云安全主要体现在哪些方面呢？这里简要介绍一些云安全所需考虑的因素。

- **数据安全：**云服务提供商需要保护云用户的数据不被窃取或丢失。强加密及密钥管理是云计算系统用以保护数据的一种核心机制。加密虽然不能保证数据不会丢失，但是对于无法获取明文的数据来说，数据被窃取的危害则显得不再那么大，有的法律法规甚至认为可以不对加密数据的丢失负有责任。密钥管理则提供了对保护资源的访问控制。Keystone 中引入令牌机制来管理用户对资源的访问，同时引入了 PKI（公钥基础实施）对令牌加以保护。
- **身份和访问管理安全：**有效的身份和访问控制是云平台中必不可少的一个环节。对于云计算中的用户和服务认证，除了基于风险的认证方法外，还需要注意简单性和易用性。云计算中需要注意合理定义系统管理人员的控制边界，以防来自内部的攻击所造成的危害。Keystone 中通过 Policy（访问规则）来做到基于用户角色的访问控制。
- **虚拟化安全：**云计算离不开虚拟化，虚拟化技术在计算能力、网络、内存等方面的应用扩展了多租户下的云服务。然而虚拟化技术也带来了一些安全问题：如何有效地安全隔离各个虚拟机，以使得数据不会被污染；不同敏感度和安全要求的虚拟机如何共存，以防止安全保护低的虚拟机成为多租户下的瓶颈；虚拟操作系统中缺少

安全保护的有效机制以及如何对虚拟机之间的通信进行安全控制。

- **基础设施安全：**基础设施安全包括服务器、存储、网络等核心 IT 基础设施的安全。这些基础设施的安全性考虑历来有之，但是在云计算的环境下，相对于自建系统来说安全性问题变得更加严重。服务器层的安全控制措施，例如强认证、安全事件日志、基于主机的入侵检测系统或入侵防御系统等；网络层面的安全控制措施，例如传输数据加密，基于网络的入侵检测系统或入侵防御系统等。可信计算池（Trusted Compute Pools）通过对计算节点的硬件以及系统内核进行度量来确定一个可信任计算节点的集合，可信计算池的引入提高了基础设置的安全性。

Keystone 作为 OpenStack 中一个独立的提供安全认证的模块，主要负责 OpenStack 用户的身份认证、令牌管理、提供访问资源的服务目录，以及基于用户角色的访问控制。用户访问系统的用户名密码是否正确、令牌的颁发、服务端点的注册，以及该用户是否具有访问特定资源的权限等都离不开 Keystone 服务的参与。

可信计算池是 Intel 提出的一个特性，管理员可以通过可信计算池来定义一组主机为可信计算节点。可信计算池得益于 Intel 可信执行技术（TXT）提供的硬件层面上的安全特性，通过信任链的建立来保证计算环境上的软硬件经过正确的度量，经过度量且可信任的计算节点将可以被加入到可信计算池中，从而满足当云用户对可信任计算环境的要求，也就是说，通过可信计算池，云用户可以将数据或者业务只部署在可信任服务器上。

8.2 Keystone

OpenStack 身份管理服务（Identity Service），即 Keystone，是 OpenStack 早期版本就独立出来的一个核心项目。在 OpenStack 的整体框架结构中，Keystone 的作用类似一个服务总线，Nova、Glance、Horizon、Swift、Cinder 以及 Neutron 等其他服务通过 Keystone 来注册其服务的 Endpoint（可理解为服务的访问点或 URL），针对这些服务的任何调用都需要经过 Keystone 的身份认证，并获得服务的 Endpoint 来进行访问。

8.2.1 Keystone 体系结构

对于 Keystone，我们首先需要的是澄清一些基本的概念。

- **Domain：**域。Keystone 中的域是一个虚拟的概念，由特定的项目（Project）来承担。一个域是一组 User、Group 或 Project 的容器，一个域可以对应一个大的机构，一个数据中心，必须全局唯一。云服务的客户是 Domain 的所有者，他们可以在自己的 Domain 中创建多个 Projects、Users、Groups 和 Roles。通过引入 Domain，云服务客户可以对其拥有的多个 Project 进行统一管理，而不必再像过去那样对每一个 Project 进行单独管理。
- **User：**用户。通过 Keystone 访问 OpenStack 服务的个人、系统抑或是某个服务，Keystone

会通过认证信息（Credential，比如密码等）验证用户请求的合法性，通过验证的用户将会分配到一个特定的令牌，该令牌可以用作后续资源访问的一个通行证，非全局唯一，只需要在域内唯一即可。

- **Group:** 用户组。一组 Users 的容器，可以向 Group 中添加用户，并直接给 Group 分配角色，那么在这个 Group 中的所有用户就都拥有了 Group 所拥有的角色权限。通过引入 Group 的概念，Keystone 实现了对用户组的管理，达到了同时管理一组用户权限的目的。
- **Project:** 项目。项目是各个服务中的一些可以访问的资源集合，例如，在 Nova 中，我们可以把项目理解成一组虚机的拥有者，在 Swift 中则是一组容器的拥有者。基于此，我们需要在创建虚拟机时指定某个项目，在 Cinder 创建卷也是要指定具体的项目。用户默认的总是绑定到某些项目上，用户访问项目的资源前，必须具有对该项目的访问权限，或者说是在特定项目下赋予了特定角色。项目不必全局唯一，只需要在某个域下唯一即可。
- **Role:** 角色。一个用户所具有的角色，角色不同意味着被赋予的权限不同，只有知道用户所被授予的角色才能知道该用户是否有权访问某资源。用户可以被赋予一个域或者项目内的角色。一个用户被赋予域的角色意味着对域内所有的项目具有相同的角色，而特定项目的角色只具有对特定项目的访问权限。角色可以继承，在一个项目树下，用于对父项目的访问权限也意味着同时拥有对子项目的访问权限。角色必须全局唯一。
- **Service:** 服务，比如 Nova、Swift、Glance、Cinder 等。根据 User、Tenant 和 Role 一个服务可以确认当前用户是否具有访问其资源的权限。服务对外暴露一个或者多个端点（Endpoint），用户只有通过这些端点才可以访问所需资源或者执行某些操作。
- **Endpoint:** 端点。端点是指一个可以用来访问某个具体服务的网络地址，因此我们可以将端点理解为服务的访问点。如果需要访问一个服务，就必须知道它的 Endpoint。一般以一个 URL 地址来表示一个端点，URL 细分为 Public、Internal 和 Admin 3 种，Public URL 是为全局提供的服务端点，Internal URL 相对于 Public URL 来说提供给内部服务之间的访问，Admin URL 提供给系统管理员使用。
- **Token:** 令牌。令牌是允许访问特定资源的凭证。无论通过某种方式，Keystone 最终的目的就是对外提供一个可以访问资源的令牌。用户可以通过 Credential 获取在某个项目下的令牌。
- **Credentials:** 凭证。用户的用户名和密码。

上述概念之间的关系如图 8-1 所示。

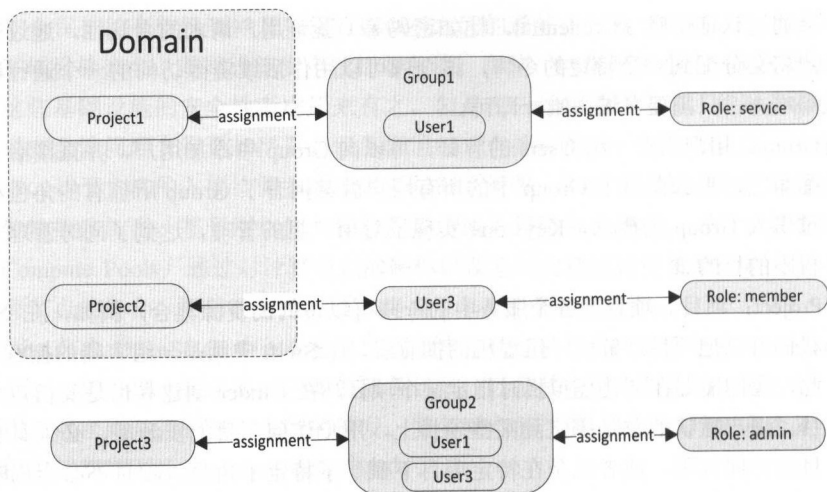


图 8-1 域、项目、用户、组与角色

基于这些核心的概念, Keystone 主要提供了 Authentication (认证)、Token(令牌)、Catalog (目录) 和 Policy (安全策略, 或者说访问控制) 4 个方面的核心服务。

- **Authentication** : 对用户的身份进行验证, 用户的身份凭证通常是以用户名和密码形式呈现, 认证服务同时提供了与该用户相关的元数据, 例如用户的项目角色。
- **Token**: 确认用户的身份之后, 会给用户提供一个核实该身份并且可以用于后续资源请求的令牌, Token 服务则验证并管理用于验证身份的令牌。Keystone 会颁发给通过认证服务的用户两种类型的令牌, 一类是无明确访问范围的令牌 (unscoped token), 此种类型的令牌存在的主要目的是用来保存用户的 credential, 可以基于此令牌获取有确定访问范围的令牌 (scoped token)。虽然意义不大, 但是 Keystone 还是保留了基于 unscoped token 查询 Project 列表的功能, 用户选择要访问的 Project, 继而可以获得与 Project 或者域绑定的令牌, 只有通过某个特定项目或者域相绑定的令牌, 才可以访问此项目或者域内的资源。令牌只在有限的时间内有效。
- **Catalog**: Catalog 服务对外提供一个服务的查询目录, 或者说是每个服务的可访问 Endpoint 列表。服务目录存有所有服务的 Endpoint 信息, 服务之间的资源访问首先需要获取到该资源的 Endpoint 信息, 通常是一些 URL 列表, 然后才可以根据该信息进行资源访问。从目前的版本来看, Keystone 提供的服务目录是与有访问范围的令牌同时返回给用户的。
- **Policy**: 一个基于规则的身份验证引擎, 通过配置文件来定义各种动作与用户角色的匹配关系。严格来讲这部分内容现在已经不是隶属于 Keystone 项目了, 这是源于访问控制在不同的项目中都有涉及, 所以这部分内容是作为 Oslo 的一部分进行开发维护。

通过这几个服务，Keystone 在用户与服务之间架起一道桥梁：用户从 Keystone 获取令牌以及服务列表；用户访问服务时，发送自己的令牌；相关的服务向 Keystone 求证令牌的合法性。下面以创建虚拟机为例，如图 8-2 所示为 Keystone 的工作流程。

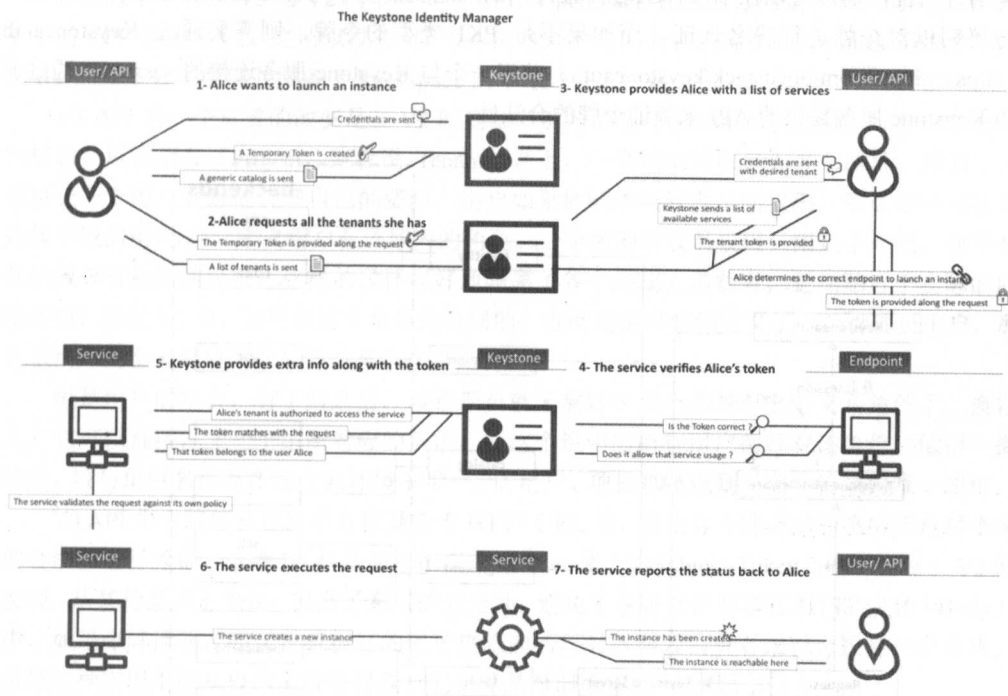


图 8-2 Keystone 工作流程

- 用户 Alice 发送自己的凭证到 Keystone，Keystone 认证通过后，返回给 Alice 一个 unscoped token 以及服务目录。
- Alice 通过 unscoped token 向 Keystone 查询当前环境下的项目列表，Keystone 验证 token 成功后，返回 Alice 一个项目列表。到此为止的操作仅是为了查询项目，如果已经知道要访问的项目，则可以忽略这两个步骤直接开始下面的流程。
- Alice 选择一个项目，发送自己的凭证给 Keystone 申请一个 scoped token，Keystone 验证后，返回 scoped token。
- Alice 凭借 scoped token 发送请求到计算服务的 Endpoint 以创建虚拟机，Keystone 验证 scoped token（包括该 token 是否有效，是否有权限创建虚拟机等）成功后，再把请求转发到 Nova，最终创建虚拟机。

1. Keystone 架构

Keystone 体系结构如图 8-3 所示，除了 keystoneclient 之外，Keystone 还涉及另外一个子

项目 keystonemiddleware (<https://github.com/openstack/keystonemiddleware>)。keystonemiddleware 是 Keystone 提供的对令牌合法性进行验证的中间件, 比如, 客户端访问 Keystone 提供的资源时提供了 PKI 类型的令牌类型, 为了不必每次都需要 Keystone 服务的直接介入以验证令牌的合法性, 通常可以在中间件上进行验证, 当然这就需要中间件上已经缓存了相关的证书与秘钥以对令牌进行签名认证。而如果不是 PKI 类型的令牌, 则需要通过 Keystoneauth (<https://github.com/openstack/keystoneauth>) 获得一个与 Keystone 服务连接的 session, 通过调用 Keystone 服务提供的 API 来验证令牌的合法性。

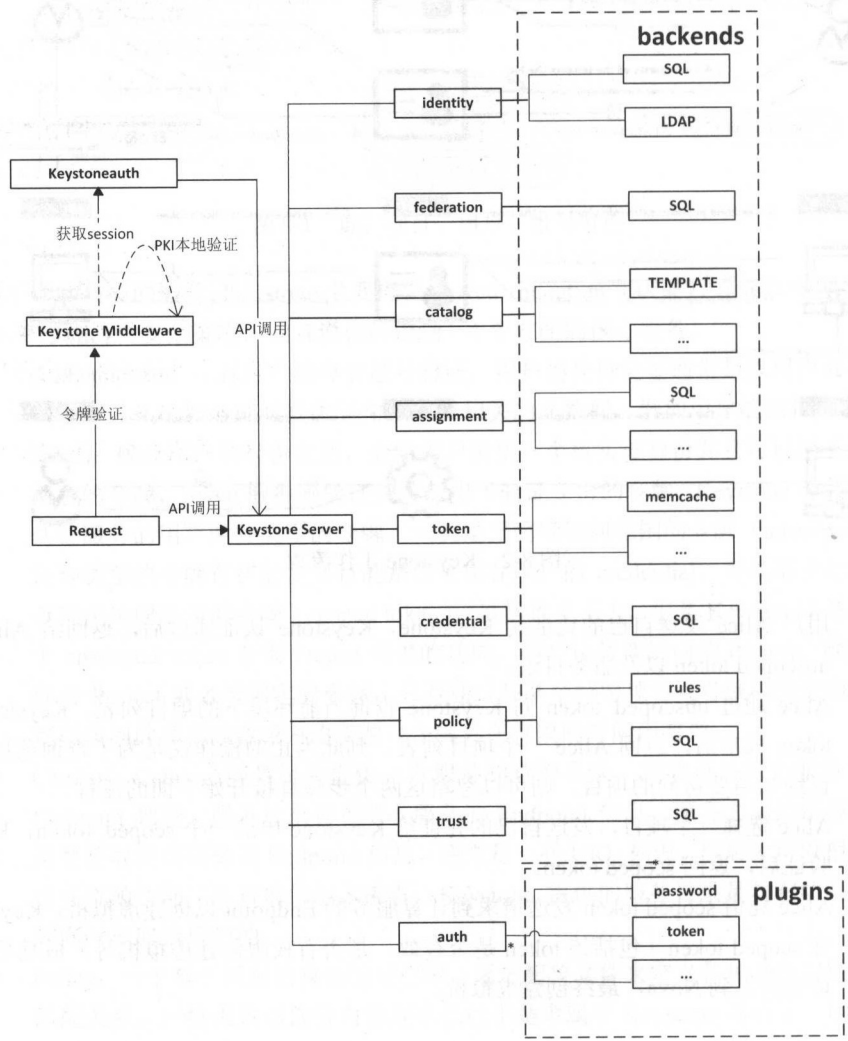


图 8-3 Keystone 架构

Keystone 项目本身，除了后台的数据库外，主要包括一个处理 RESTful 请求的 API 服务进程。这些 API 涵盖了 Identity、Token、Catalog 和 Policy 等 Keystone 提供的各种服务，这些不同服务所能提供的功能则分别由相应的后端 Driver（Backend Driver）实现。

目前版本中存在 V3 与 V2 两个版本的 API，OpenStack 社区的方向是在不久的将来用 V3 版本的 API 来取代 V2。之所以没有 V1 版本的痕迹，是因为它出现在 OpenStack 诞生之前，由 Rackspace 公司实现并服务于 Rackspace 的早期公有云产品。

V3 API 的一个重要的改变是在 V2 的基础上引入了域（Domains）以及用户组（Groups）的概念。域在项目（Project，或者说 Tenant）之上，一个域中可以包含多个项目。域的引入可以让一个用户更好地管理自己的资源。用户如果被赋予域管理员的权限，那么他就可以创建属于域的用户/组，定义用户在该域内的角色。一个域的管理员权限只限定于此域，他不对其他域享有相同的权限，这样的设计很好地隔离了各个终端云消费者，更加贴近于实际的应用环境，而在 V2 中，管理员这个角色是全局的，也就是说只要你定义了一个管理员用户，那么该用户是全局有效而非只针对某个项目而言。

组是用户的集合，有了组之后，域管理员就无需针对单个的用户来定义其角色了，换言之，它可以直接定义一个组所对应的角色，而这个组中的所有用户都具有该角色。值得一提的是，域与角色名需要在这个云环境下唯一，而用户、项目以及组则只需在该域内唯一即可。

V3 API 中令牌信息可以不直接暴露于 HTTP URL 中，无论对令牌请求的响应还是对令牌的合法性进行验证，令牌 ID 都是保存在请求 Header 域“X-Subject-Token”中。相对于 V2 的实现，从某种意义上来说，提高了系统的安全性，避免了令牌直接暴露在 HTTP 主体（Body）中。如果令牌泄露，而又没有很好的保护措施的话，对终端云消费者来说无疑是一个灾难。虽然这种实现不能从根本上解决问题，但也算是对旧版本的一个优化。

2. Keystone 源码结构

```
.
├── api-ref - 唯一可信 API 定义文档
├── doc
├── etc
│   ├── default_catalog.templates - 服务目录模板文件
│   ├── keystone.conf.sample - 主配置文件
│   ├── keystone-paste.ini - Paste Deploy 配置文件
│   ├── logging.conf.sample
│   ├── policy.json - 访问控制配置文件
│   ├── policy.v3cloudsample.json - 支持多 domain 访问控制配置文件
│   └── sso_callback_template.html
├── examples - 提供 PKI 初始化所写的一些证书及密钥，以及生成这些文件的示例脚本
├── httpd - 配置 Keystone 与 Apache HTTPD 服务一起启动的一些示例文件
├── keystone
│   ├── assignment - 用户角色授权
│   └── auth - 用户认证模块
```

```

|   |—— catalog - 提供一个可以访问的服务目录
|   |—— cmd - 命令行支持
|   |—— contrib - 扩展的 API 实现
|   |—— credential - 用户密钥管理
|   |—— endpoint_policy - 基于 endpoint 的 policy 管理
|   |—— federation - 提供联合身份管理
|   |—— identity - 用户身份管理
|   |—— middleware - WSGI 中间件
|   |—— oauth1 - 提供对 OAuth1 支持
|   |—— policy - 用户自定义 Policy 配置
|   |—— resource - 管理项目和域
|   |—— revoke - 回收消息管理
|   |—— token - 令牌管理模块
|   |—— trust - 提供访问权限代理
|   |—— v2_crud - 已弃用的 V2 的 CRUD 操作
|   |—— version - 当前 API 版本信息
|—— setup.cfg
|—— setup.py
|—— test

```

keystone 目录下并没有其他项目那样有一个专门的 api 子目录来针对各种 Keystone API 进行实现，而是基本上由针对 Identity 等不同服务的子目录组成，而且这些子目录下的代码结构也基本保持一致。下面以 Catalog 服务为例：

```

.
|—— backends
|   |—— base.py
|   |—— sql.py
|   |—— templated.py
|—— controllers.py
|—— core.py
|—— routers.py
|—— schema.py

```

基本上都会有 3 个文件，routers.py 定义路由规则，通过这些规则，可以将每个 API 请求路由到 controllers.py 中定义的具体 Controller；core.py 则定义了 Manager 类与 Driver 类，Manager 负责基于不同的 Backend Driver 对请求进一步处理。Driver 层与 Manager 层之间的关系如图 8-4 所示。

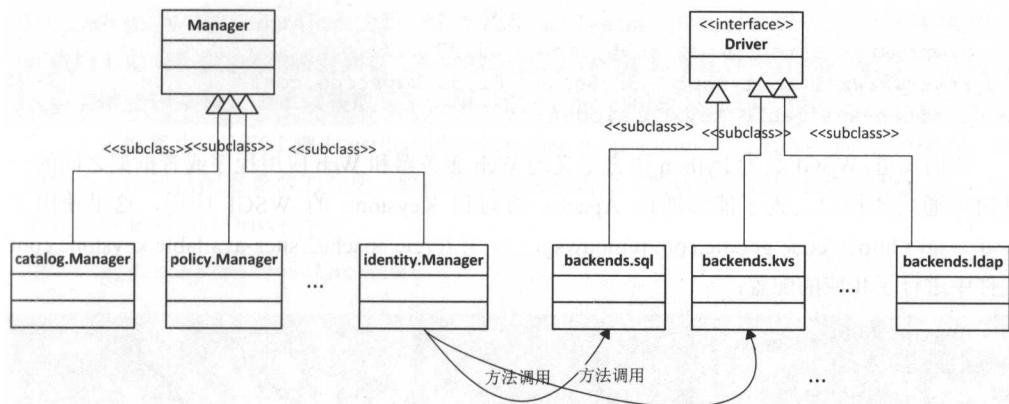


图 8-4 Driver 与 Manger 关系

各种 Backend Driver 代表着不同的后端实现方式，比如图 8-3 中所示的 SQL、KVS (Key-Value Store)、LDAP 等，其他在 Keystone 中还用到的有 Template（可以理解为一种特殊的 KVS 实现）、MemCache（高速缓冲存储系统）等。

- SQL：利用 SQLAlchemy 做数据的持久化。
- KVS：通过主键查询的方式可以极大地支持海量数据存储，被广泛应用于缓存、搜索引擎等领域。Keystone 中 KVS 则主要结合缓存来实现数据的存储。
- LDAP：轻量目录访问协议，以树状的层次结构来存储数据。
- Template：主要用在服务目录中，通过模板的方式支持用户自定义一个当前系统环境下可用的服务目录。

8.2.2 Keystone 启动过程

传统上 Keystone 是通过 bin/keystone-all 脚本进行启动的。这种启动方式基于 Eventlet，但是因为 Keystone Federation（联合身份管理）依赖于 Apache 的支持，例如对 SAML 文件进行校验，以及在处理多线程时 Apache HTTPD 服务较于 Eventlet 的优势，Keystone 已经在 Newton 版本中不再对 Eventlet 部署方式提供支持。

使用 Devstack 进行 OpenStack 部署时，默认是采用 Apache/mod_wsgi 的方式进行部署。在新版本中，除了采用 mod_wsgi 外，还支持 Apache/mod_proxy_uwsgi 的方式进行部署。源码中均有支持两种部署的默认配置文件。

Keystone 是作为 Apache 的一个模块随 Apache 服务的启动而启动的，此时，使用 screen 进行开发调试时，我们就需要通过重启 Apache 服务来运行新的 Keystone 代码。

下面以 Ubuntu 为例，Apache 下 Keystone 的核心配置文件位于：

```
$ ll /etc/apache2/sites-enabled
```

```
total 8
drwxr-xr-x 8 root root 4096 Sep  9 23:27 ../
lrwxrwxrwx 1 root root   32 Sep 10 01:56 keystone.conf
-> ../sites-available/keystone.conf
```

我们知道, WSGI 是为 Python 语言定义的 Web 服务器和 Web 应用程序或者框架之间的一种简单通用的接口。为了能够通过 Apache 访问到 Keystone 的 WSGI 应用, 这里使用了 `mod_wsgi` (<https://code.google.com/p/modwsgi/>), 并在 `/etc/apache2/sites-available/keystone.conf` 文件中进行了相应的配置:

```
Listen 5000
Listen 35357
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-agent}i\" %D(us)" keystone_combined

<VirtualHost *:5000>
    WSGIDaemonProcess keystone-public processes=5 threads=1 user=stack
    display-name=%{GROUP}
    WSGIProcessGroup keystone-public
    WSGIScriptAlias / /usr/local/bin/keystone-wsgi-public
    WSGIApplicationGroup %{GLOBAL}
    WSGIPassAuthorization On
    ErrorLogFormat "%M"
    ErrorLog /var/log/apache2/keystone.log
    CustomLog /var/log/apache2/keystone_access.log keystone_combined
</VirtualHost>

<VirtualHost *:35357>
    WSGIDaemonProcess keystone-admin processes=5 threads=1 user=stack
    display-name=%{GROUP}
    WSGIProcessGroup keystone-admin
    WSGIScriptAlias / /usr/local/bin/keystone-wsgi-admin
    WSGIApplicationGroup %{GLOBAL}
    WSGIPassAuthorization On
    ErrorLogFormat "%M"
    ErrorLog /var/log/apache2/keystone.log
    CustomLog /var/log/apache2/keystone_access.log keystone_combined
</VirtualHost>
```

这里开启了两个 TCP 的访问端口 5000 与 35357。按照官方的解释, 5000 是公用端口, 35357 则是管理端口。之所以有 5000 这个服务端口是有其历史原因的, 早在 V3 API 出现之前, Keystone 定义了一些只有管理员才可以调用的 API, 例如修改用户密码这种操作。而这些操作是不对普通用户开放的, 所以才来有了这两个服务端口。但是在 V3 API 出现出后, 这样的操作完全可以有 Policy 去管控, 因此也就显得有点多余了。与之对应的是开启了两个独立的

VirtualHost。WSGIScriptAlias 的第一个参数是 URL-path，对应这里的值为“/”，代表以“/”开头的 URL 路径将会被映射到第二个参数指定的 WSGI 脚本文件中，mod_wsgi 在启动的时候会调用到该 WSGI 脚本以获取一个应用对象（Application Object）。

上述配置中，WSGI 脚本/usr/local/bin/keystone-wsgi-admin 就是 Keystone 启动脚本，所做的核心工作包括配置参数的注册、数据库连接的初始化（默认的数据库引擎为 SQLite），以及加载各种 Backend Driver 等。

```
# keystone/server/backends.py

# 加载 Backend Driver
def load_backends():

    cache.configure_cache()
    cache.configure_cache(region=catalog.COMPUTED_CATALOG_REGION)
    ...

IDENTITY_API = identity.Manager()
ASSIGNMENT_API = assignment.Manager()

DRIVERS = dict(
    assignment_api=_ASSIGNMENT_API,
    catalog_api=catalog.Manager(),
    credential_api=credential.Manager(),
    credential_provider_api=credential.provider.Manager(),
    domain_config_api=resource.DomainConfigManager(),
    endpoint_policy_api=endpoint_policy.Manager(),
    federation_api=federation.Manager(),
    id_generator_api=identity.generator.Manager(),
    id_mapping_api=identity.MappingManager(),
    identity_api=_IDENTITY_API,
    shadow_users_api=identity.ShadowUsersManager(),
    oauth_api=oauth1.Manager(),
    policy_api=policy.Manager(),
    resource_api=resource.Manager(),
    revoke_api=revoke.Manager(),
    role_api=assignment.RoleManager(),
    token_api=token.persistance.Manager(),
    trust_api=trust.Manager(),
    token_provider_api=token.provider.Manager())

auth.controllers.load_auth_methods()
return DRIVERS
```

在“DRIVERS”字典中，每一个键值对都定义了一类 Keystone API 的实现，它们之间存在着相互依赖的可能，比如 Assignment API 需要用到 Credential API 以及 Identity API，其他的 API 也有可能需要用到 Assignment API。为了解决这种相互依赖的问题，Keystone 通过 `keystone.common.dependency.resolve_future_dependencies()` 方法来解决。

至于后续的启动过程，与 Nova 等其他项目并没有多大的差别，可以参考前面章节的介绍。

8.2.3 用户认证及令牌获取

一个有效的 curl 请求如下，该请求对用户的账号进行验证并生成一个与项目进行绑定的令牌。

```
$ curl -i -H "Content-Type: application/json"
-d '{
  "auth": {
    "identity": {
      "methods": ["password"],
      "password": {
        "user": {
          "name": "admin",
          "domain": { "id": "default" },
          "password": "password"
        }
      }
    },
    "scope": {
      "project": {
        "name": "admin",
        "domain": { "id": "default" }
      }
    }
  }
} \
http://localhost:5000/v3/auth/tokens
```

经过路由，此 API 请求对应的 Controller 为 `keystone.auth.controller.Auth`，Action 为 `authenticate_for_token()` 方法：

```
class Auth(controller.V3Controller):
    def authenticate_for_token(self, request, auth=None):
```

第二个参数 `auth` 就是我们的 curl 请求中的传入参数，即为此项目下的用户名称以及密码。`authenticate_for_token()` 首先针对 Keystone 的 3 种认证方式分别进行处理。

- 基于令牌：如果令牌包含在参数 `auth` 中，则通过令牌信息来完成认证，剥离 HTTP

请求 Header 中的令牌信息，进行哈希计算并与数据库中保存的令牌值进行比对以确认是否有效。令牌在此的作用等效于用户名和密码。

- 外部用户：如果上下文信息 context 中包含外部用户“REMOTE_USER”信息，认证该外部用户的关联项目以及角色的合法性，并用自定义的方式进行认证，例如 Kerberos。
- 本地认证：默认方式，这里的例子为本地认证的方式，即验证用户名与密码。本地认证核心的操作是通过 Backend Driver 去做密码的校验，就是对传入明文做一次 SHA512 的哈希操作，再与数据库中存放的散列之后的密码进行比对。

此外还有 OAuth1 方式，一种通过 OAuth1 协议实现访问权限代理的认证方式，以及以 mapped 认证方式对 federation 功能提供支持，从 M 版本开始还支持基于时间的一次性密钥的认证方式（TOTP），但是目前版本还不提供对多因素认证方式的支持（multifactor authentication）。

校验完成之后会接着检查该用户、域、项目是否可用，过滤掉不需要返回给用户的数据，并调用 Catalog API 构造服务目录，最后就是通过具体的 Backend 去生成令牌。令牌可以有 4 种生成方式，默认的是使用 UUID 的方式，之后的版本将会由 Fernet 令牌取代：

- UUID：调用 Python 库函数来生产一个随机的 UUID（通用唯一识别码）作为令牌的 ID。
- PKIZ：使用 OpenSSL 对用户相关信息进行签名，签名后的格式为 DER，并以此来生成令牌 ID，从 M 版本开始不建议使用，后续版本会删除对 PKIZ 令牌的支持。
- PKI：使用 OpenSSL 对用户相关信息进行签名，与 PKIZ 不同的是生成的签名格式为 PEM，从 M 版本开始不建议使用，后续版本会删除对 PKIZ 令牌的支持。
- Fernet token：基于 Fernet（一种对称加密算法）的对用户信息进行加密而生成的令牌，与用户相关的所有的认证信息都保存在令牌中，故令牌本身就包含了所有的信息，也因此 Fernet 令牌无需持久化，而一旦一个 Fernet 令牌被攻破，那么他就可以做所有该用户能做到的事情。

到此为止的内容涵盖了 Keystone 的几个核心功能，包括对外提供访问目录以及令牌服务等。本节开始提供的示例所返回的内容如下：

```
# 令牌
X-Subject-Token: d1802da99abe43038b62cd9732c6d048
Vary: X-Auth-Token
x-openstack-request-id: req-1fbd1ee4-8dec-4227-aale-46c2e090b5c8
.....
{
  "token": {
    "is_domain": false,
    "service_providers": .....,
    "methods": ["password"],
    "roles": [{
      "id": "9cf8420ea5324f79b9d740e3ce5f0e04",
      "name": "admin"
```

```

    },
    "is_admin_project": true,
    "project": {
        "domain": {
            "id": "default",
            "name": "Default"
        },
        "id": "caaa06cf868e44c2882623afd34eea60",
        "name": "admin"
    },
    # 服务目录，提供这个服务对应的可访问端点列表
    # 计算服务的可访问端点列表
    "catalog": [{
        "endpoints": [{
            "region_id": "RegionOne",
            "url":
"http://10.239.159.68:8774/v2/caaa06cf868e44c2882623afd34eea60",
            "region": "RegionOne",
            "interface": "public",
            "id": "7637807716e541dea65679c07900657d"
        }, {
            "region_id": "RegionOne",
            "url":
"http://10.239.159.68:8774/v2/caaa06cf868e44c2882623afd34eea60",
            "region": "RegionOne",
            "interface": "admin",
            "id": "c0f96b735e394b319e156cf466184fcc"
        }, {
            "region_id": "RegionOne",
            "url":
"http://10.239.159.68:8774/v2/caaa06cf868e44c2882623afd34eea60",
            "region": "RegionOne",
            "interface": "internal",
            "id": "ebl06163e6941738f0b543f983320d0"
        }
    ],
    "type": "compute_legacy",
    "id": "13e6ef6f64f940aab31b3346fc4d5b9e",
    "name": "nova_legacy"
    .....
    # 身份管理服务的可访问端点列表
    }, {
        "endpoints": [{
            "region_id": "RegionOne",
            "url": "http://10.239.159.68/identity",

```

```

        "region": "RegionOne",
        "interface": "public",
        "id": "4387c5fcddfb4bec8d4cf40da835c0f1"
    }, {
        "region_id": "RegionOne",
        "url": "http://10.239.159.68/identity",
        "region": "RegionOne",
        "interface": "internal",
        "id": "a4ceb047cdac40219997c0eb7d34c771"
    }, {
        "region_id": "RegionOne",
        "url": "http://10.239.159.68/identity_v2_admin",
        "region": "RegionOne",
        "interface": "admin",
        "id": "cb6520ebdfb64c8db552c8bc0dcd4ff7"
    }],
    "type": "identity",
    "id": "6a07c77a04df4e88b0d3f5a0f11a7fbe",
    "name": "keystone"
    .....
}],
"expires_at": "2016-09-16T16:10:48.691237Z",
"user": {
    "domain": {
        "id": "default",
        "name": "Default"
    },
    "id": "48ea5e01709d4eb0aa77bdd5c288cdfe",
    "name": "admin"
},
"audit_ids": ["wuSpV8nhRmiwzLOWctk0RA"],
"issued_at": "2016-09-16T06:10:48.691261Z"
}
}

```

8.2.4 签名证书生成

下面介绍 Keystone 在如何生成签名证书之前，需要了解的一些基础的概念。

- 权威认证机构 (Certificate Authority): 一般简称为 CA，是 PKI 的核心组成部分，也称作认证中心。它是数字证书的签发机构。CA 是 PKI 应用中权威的、可信任的、公正的第三方机构。
- CA 私钥: 非对称加密算法中密钥分为公钥和私钥，公钥可对外暴露，私钥则需要对

外保密。简单来说，私钥可以用来签名和解密，公钥则用来解签与加密。而 CA 私钥必须绝对保密，因为它是信任链的根，如果 CA 私钥泄露，整个 PKI 系统也就变得毫无意义。

- CA 公钥证书：CA 公钥证书中包含了 CA 的公钥信息，可以用 CA 公钥证书对 CA 颁发的证书进行签名验证，以确认 CA 机构颁发证书的合法性。
- 签名私钥：一般为用户自己负责保存的一份私钥，签名私钥不能泄露，可以用来对传输数据进行签名，以保证数据的合法性。
- 签名公钥证书：由 CA 机构颁发，签名证书上有 CA 的签名以表明该签名证书的合法性，同时证书中保存的公钥（公钥与签名私钥为一对非对称秘钥对）可以用来对数据进行解签。

在 Folsom 版本以前，生成令牌 ID 的方法只有 UUID 这种方式。生成的令牌保存在 Keystone 的后台数据库中，同时通过网络传给客户端，客户端有了令牌 ID 之后，每个请求将会包含这一信息。Keystone 在拿到这些信息之后将会提取令牌 ID 和后台数据库中的值进行比较以验证请求的合法性。

这样做的问题在于，在大规模的集群环境中，同时有大量客户端并发请求的情况下，Keystone 的性能将会是一个大的瓶颈。因为每个请求都需要和 Keystone 进行交互以对令牌的合法性进行校验；除此之外，如果令牌被无意泄露或者窃取，也将会是一个问题。

于是 PKI 系统在之后的版本中被引入，也就是之前所说的 Keystone 会利用 PKI 对令牌相关的数据进行签名，而每一个服务端点会保存一份签名公钥证书、CA 公钥证书以及证书吊销列表，这样，即可以进行本地校验而无需与 Keystone 频繁交互。整个过程如图 8-5 所示。

- 客户端发送用户名密码信息到 Keystone 进行验证。Keystone 校验用户名密码以及项目信息合法之后用签名私钥对令牌元数据（元数据具体为该令牌颁发给的用户，所在的项目，在项目下的用户角色以及该令牌可见的服务目录等信息）进行签名以生成令牌。
- 令牌通过网络发送到客户端，令牌同时缓存在客户端。
- 客户端发送 API 请求到服务端点，服务端点提取令牌信息，用本地存放的签名公钥证书进行验签。
- 服务端点处理合法的请求，拒绝验证未通过的请求。

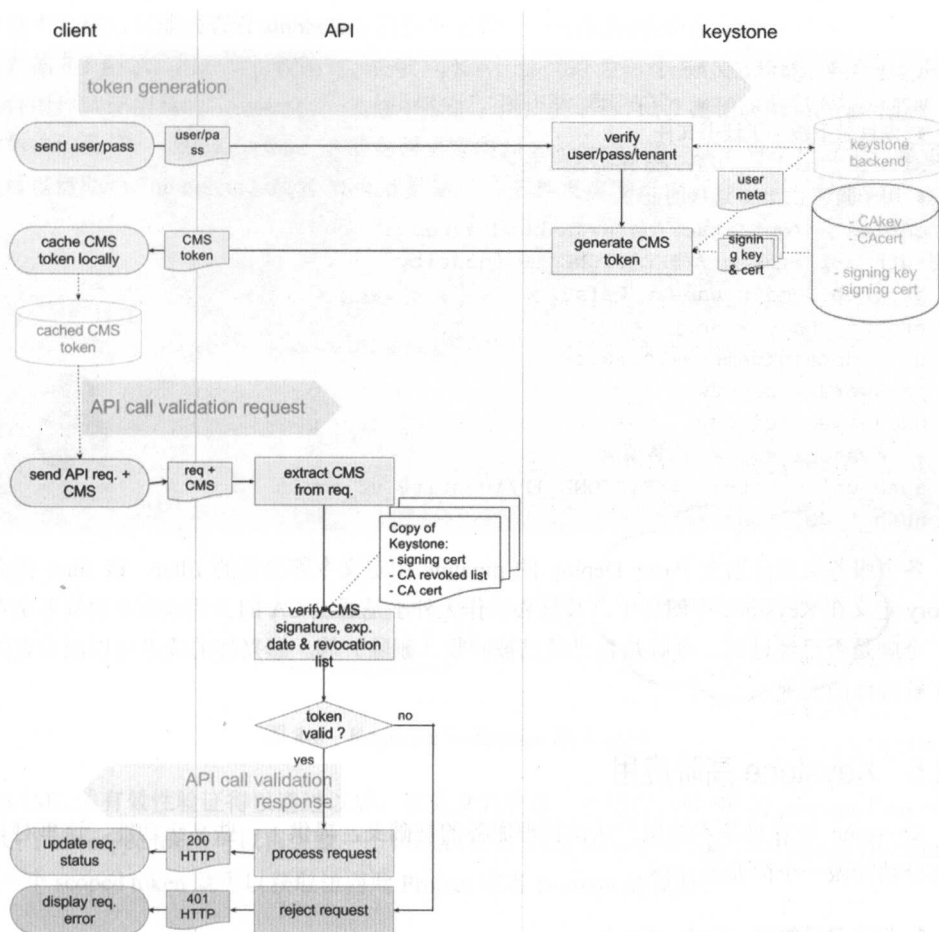


图 8-5 PKI 签名过程

假设用户的系统之前没有设置过 PKI 环境,可以运行下面的命令,来生成证书以及秘钥。

```
$ keystone-manage pki_setup
```

由此触发的操作位于 `keystone.cmd.cli.PKISetup`, 其中的注释明确说明不建议用在正式的商业环境中。这个命令设置的 PKI 环境只是为了概念验证, 因为很少会有哪个产品会冒这样的风险去搭建自己的 PKI 系统, 正如我们前面所说的做到 PKI 的权威性、可信任性以及公正性是一个系统工程, 事实上, 大部分企业宁愿去外部购买 PKI 服务。

令牌的签名验证工作在各个服务端点上, 这就要求各个服务端点上对相关的配置进行定义, 例如配置认证 URL、认证主机以及端口等信息, 当作令牌认证时, 这些配置会在 Keystone 中间件 (`keystonemiddleware`) 项目中给予加载。比如 Cinder 的主配置文件中配置如

下:

```
[keystone_authtoken]
memcached_servers = $CACHE_SERVER_IP:11211
# 该目录下缓存了证书文件
signing_dir = /var/cache/cinder
# 用于验证 HTTPS 连接的证书
cafile = /opt/stack/data/ca-bundle.pem
auth_uri = http://$KEYSTONE_IP/identity
project_domain_name = Default
project_name = service
user_domain_name = Default
password = zaql2wsx
username = cinder
# Keystone server 服务端点
auth_url = http://$KEYSTONE_IP/identity_v2_admin
auth_type = password
```

各个服务端点分别在 Paste Deploy 的 pipeline 中定义令牌验证的 filter。该 filter 指向的 factory 定义在 Keystone 中间件中, 其核心工作无外乎是通过 CA 的公钥验证令牌签名是否有效、令牌是否已经过期、令牌是否已经已被回收(删除)等。感兴趣的读者可以浏览它的代码了解具体的实现。

8.2.5 Keystone 高阶应用

Keystone 在提供基本的用户认证管理服务的基础上, 提供了一些高级功能, 这里对其中的部分功能做一个简要的介绍。

1. 联合身份管理 (Federation)

联合身份管理的目的在于将身份认证部分单独剥离出来, 即有一个独立的节点负责对用户身份进行认证 (IdP), IdP 认证通过的用户可以访问服务提供商 (SP) 所提供的服务。在对 IdP 完全信任的基础上, 可以实现对资源的安全访问。用户可以用一个密码访问多个可被授权访问的云系统, 用户也无需重新登录以访问不同的云系统资源。在云计算中实现联合身份管理的优点是可见的。其最大的优点是减少了维护多个云系统下多个密码的负担, 以及分散的多个密码重复登录所带来的安全性问题。

Keystone 目前支持 Keystone-Keystone Federation 以及通过设置 Keystone 和 Horizon 实现单点登录。实现 K-K Federation 需要 Keystone 运行在 Apache 下, 因为在使用 SAML 协议时, Federation 依赖于 Apache 对 SAML 文件进行校验。Keystone Federation 支持 SAML 和 OpenID Connect 两种协议。部署 Keystone-Keystone Federation 难点在于配置, 例如在配置 shibboleth2.xml 配置文件时, 一定要注意 entityID 与数据库中配置的一致性, 如果一不小心某

些配置不正确, 只能去查看 shibboleth 的日志文件, 一步步去调试了。

如图 8-6 所示, 用户身份校验时, 需要向 Keystone IdP 发一个 SAML 请求, IdP 会返回一个包含用户认证信息的 SAML 文件, 文件中描述了用户的名称, 以及在相应的 Project(Domain) 中的角色信息等。而从 Keystone SP 的访问重定向到 Keystone IdP, 从而对用户登录信息进行验证可以借助于 Shibboleth, 或者 Mellon 等第三方工具来实现。

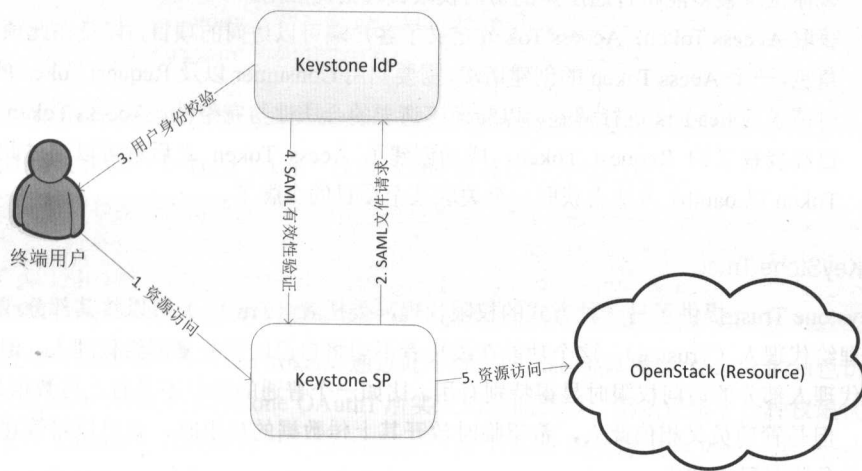


图 8-6 Keystone Federation 基本流程

SAML 的有效性验证得以通过以后, 首先拿到的是一个可以访问 SP 的 unscoped token, 通过这个 unscoped token 可以再去拿一个 Project 列表。绑定 unscoped token 和 Project(Domain) 获得一个 scoped token 就可以获取访问此 Project 或者 Domain 的权限。

2. Keystone OAuth1

在遵循 OAuth1 1.0 SPEC (<https://oauth.net/core/1.0a/>) 的基础上, Keystone 实现了将用户的访问权限代理给第三方 Consumer 的功能, 其核心理念就是不泄露访问服务提供者的密钥的前提下将部分功能开放给第三方访问者。

Keystone 中的实现可以分为 4 个部分, 即创建 Consumer、创建 Request Token、对 Request Token 进行授权, 以及获取 Access Token。

- 创建 Consumer: Keystone 对需要访问资源的客户端创建一个 Consumer, 一个 Consumer 代表了一个第三方的应用, Consumer 的 ID 对应 OAuth1 SPEC 中的 Key, secret 为 Keystone 服务和第三方应用共享的密钥, 需严格保密。此密钥用来对随后的 Access Token 以及 Request Token 请求进行签名验证。
- 创建 Request Token: 一个 Request Token 代表了一个客户端的 token 请求, 该请求只有被授权之后才能获取一个有效的 Access Token 请求。此请求必须指明需要访问的

Project，以便对其进行授权时明确其访问权限的范围（Scope）。客户端请求需要用 HMAC-SHA1 算法对 headers 进行签名，服务器同时需要通过 Consumer 的 secret 对其进行解签。

- 对 Request Token 进行授权：对一个客户端的应用必须明确其访问权限，如果你有管理员的权限，你可以将任何权限代理给第三方。但是假设你只是一个普通用户，那么你也最多能将普通用户的访问权限代理给此应用。
- 获取 Access Token：Access Token 定义了客户端可以访问的项目，以及在此项目中的角色，一个 Access Token 的创建请求，需要利用 Consumer 以及 Request Token 的 secret 对请求的 headers 进行解签，以验证该请求的合法性与完整性。Access Token 是一个已经授权了的 Request Token，成功创建了 Access Token 之后就可以通过此 Access Token 以 oauth1 方法去获取一个关联某个项目的令牌了。

3. Keystone Trust

Keystone Trust 提供了另一种方式的权限代理，委托者（Trustor）可以将其部分或者全部权限代理给代理人（Trustee）。这个功能在委托者不想将自己的账号暴露给代理人，但是又可以赋予代理人部分的访问权限时显得特别有用。比如一个普通的用户不具有上传数据到 Swift 的权限，但是管理员又相信此人，希望临时放开其上传数据的权限时，就可以将管理员上传数据的角色临时赋予此人。

Trust 可以指定一个失效日期，只有在此日期之前才有效。通过 remaining_uses 可以限定获取 Trust Token 的次数上限，代理人可以将其获得的权限再代理给其他人，也就是可以有一个 Trust Chain，Chain 的深度可以通过 redelegation_count 来限定。下面来看看一个具体的例子，假设有两个用户，Trustor 与 Trustee，Trustee 只有在 demo 项目里的访问权限，而 Trustor 则具有在 admin 项目的访问权限。我们来试试用 Trustee 在 demo 项目里的令牌去获取用户列表会得到什么。

```
$ curl -g -i -X GET http://$ENDPOINT:5000/v3/users -H "Accept: application/json" -H "X-Auth-Token: $TRUSTEE_TOKEN"

{"error": {"message": "You are not authorized to perform the requested action: identity:list_users", "code": 403, "title": "Forbidden"}}
403, Forbidden!
```

我们将 Trustor 的 admin 权限临时放开给 Trustee，通过创建一个 Trust 来实现。

```
$ curl -g -i -X POST -H "Accept: application/json" -H "X-Auth-Token: $TRUSTOR_TOKEN" -H "Content-Type: application/json" -d '{
  "trust": {
    "expires_at": "2017-02-27T18:30:59.999999Z",
    "impersonation": true,
    "allow_redelegation": true,
```

```

    "project_id": $ADMIN_PROJECT,
    "roles": [
        {
            "name": "admin"
        }
    ],
    "trustee_user_id": $TRUSTEE_USER_ID,
    "trustor_user_id": $TRUSTOR_USER_ID,
    "redelegation_count": 3
}
}' http://$ENDPOINT:5000/v3/OS-TRUST/trusts

```

现在再去获取一个 Trust Token，注意 scope 以及认证方法的定义。

```

$ curl -i -d '{"auth": {"identity": {"methods": ["token"], "token": {
{"id": "$TRUSTEE_TOKEN"} }}, "scope": {"OS-TRUST:trust": {"id":
$TRUST_ID} } } }' -H "Content-type: application/json"
http://$ENDPOINT:5000/v3/auth/tokens

```

如此便可以得到一个 Trust Token，通过此令牌，Trustee 可以模仿 Trustor 的角色访问用户列表。Keystone Trust 和 Keystone OAuth1 所实现的功能类似，都是提供了一种权限代理的方式，这里的 Trustee 可以理解为一个 Consumer，而 Trustor 则实际上为 Resource Owner，即对应 OAuth1 中对 Request Token 授权的角色。

8.3 可信计算池

在云计算环境中，可能有成千上万的计算节点部署在不同的地方，对安全等级要求高的云租户会要求其应用或虚拟机必须运行在验证为可信的计算节点之上，来保证运行环境的可信。为了满足这种需求，Intel 于 2011 年在 OpenStack Essex 版本时提出了可信计算池的特性，并将基于 Intel TXT（可信执行技术）和 OpenAttestation（OAT）远程认证项目的实现添加进了 Folsom 版本。可信计算池不断地得到来自社区的反馈，从而在后续的 OpenStack 版本中得到了来自 Intel 及社区的不断增强和优化。

8.3.1 体系结构

可信计算池的实现位于 Nova 项目，通过 FilterScheduler（过滤调度器）中加入一个新的过滤器 TrustedFilter，并与外部的一个认证服务进行交互来挑选出所有可信的主机，从而找到满足客户可信需求的目标主机环境来创建用户实例。图 8-7 所示描述了可信计算池的体系结构。

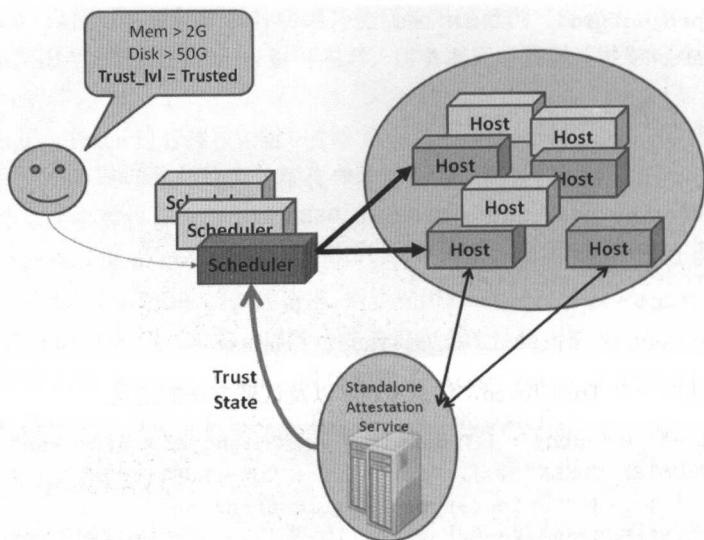


图 8-7 可信计算池体系结构

- 用户通过在 flavor 中加入新的 trust:trusted_host 键值来表示对可信主机的需求，然后通过 Nova API 发送虚拟机创建请求给调度器。
- 调度器中的 TustedFilter 从虚拟机创建请求中读取相应 flavor 的 trust:trusted_host 键值，如果是“trusted”就通过认证服务找到所有可信主机的列表，供调度器从中选择最终创建用户实例的目标主机。
- 主机（计算节点）采用基于 Intel TXT 的 TBoot 进行可信启动，对主机的 BIOS、VMM 和操作系统进行完整性度量，并在得到来自认证服务的请求时将度量数据发送给认证服务。
- 认证服务器部署基于 OAT 的认证服务，通过将来自主机的度量值与白名单数据库进行比对来确定主机的可信状态。

8.3.2 Intel TXT 与 TBoot

Intel TXT 技术首先出现在 Intel 博锐系列商用台式机平台上，现在已经扩展到服务器平台。它可以帮助防御通常相对较难应付的软件攻击，比如：

- 尝试插入不可信的虚拟机监控器（rootkit）。
- 专门用于获取内存中秘密信息的重置攻击。
- BIOS 及固件更新攻击等。

基于 TXT 的可信启动可以帮助加强对平台的控制：

- 在启动过程启用隔离和检测篡改。

- 补充运行时保护。
- 降低支持和补救成本。
- 通过基于硬件的信任增加合规保证。
- 为安全和策略应用提供信任状态来控制工作负载。

基于 TXT 的平台需要集成如图 8-8 所示的各种组件才能提供可信启动。

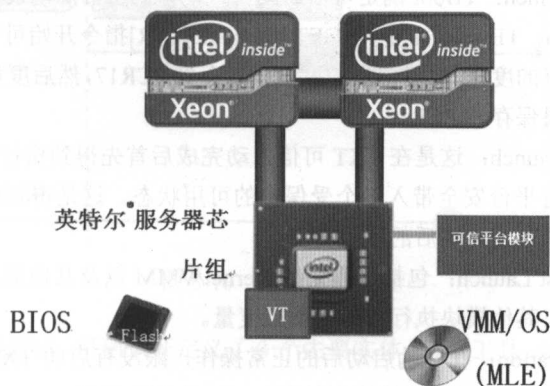


图 8-8 Intel TXT 组件

这些组件包括：

- 支持 TXT 技术和虚拟化技术的处理器。
- 支持设备虚拟化技术的芯片组。
- 可信平台模块（TPM）。
- 支持 TXT 和 TPM 的 BIOS，其中必须包含 Intel 专门为 TXT 编写并签名的已验证代码模块（ACMs）。
- 支持 TXT 的操作系统和 VMM。

基于 TXT，Intel 于 2007 年推出了一个开源项目 Trusted Boot (TBoot)，提供一个在 Kernel 和 WMM 之前执行的模块，实现 Kernel 和 VMM 的可信启动。TBoot 可信启动的流程如图 8-9 所示。

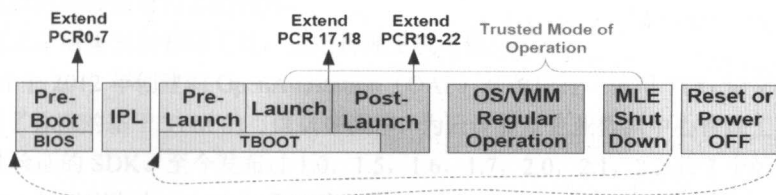


图 8-9 TBoot 可信启动流程

- **Pre-boot:** 系统固件 (BIOS/UEFI) 执行阶段。这个阶段的目标之一就是平台初始化为支持可信启动的状态。固件度量静态可信根和其他平台组件并将度量值保存到 TPM 的平台配置寄存器 (PCRs) 0 到 7 中。这一阶段还会保护 TXT 的相关资源并锁定平台的配置。
- **IPL:** 一般情况下 Kernel 执行之前的启动过程。
- **TBoot Pre-Launch:** TBoot 确定可信启动可行, 并为可信启动设置好平台状态。
- **TBoot Launch:** TBoot 通过执行 GETSEC [SENTER] 指令开始可信启动进程, 同时开始动态可信链的度量, 将可信根的度量值保存到 PCR17, 然后度量 TBoot Post-Launch 代码并将结果保存到 PCR18 中。
- **TBoot Post-Launch:** 这是在 TXT 可信启动完成后首先得到执行的 TBoot post-launch 代码, 它会将平台安全带入一个受保护的可用状态。这是得到度量的第一段系统代码, 它开启了可信启动后的度量链。
- **OS/VMM Post Launch:** 包括操作系统 Kernel/VMM 以及其他需要装载模块的启动。Kernel 负责在其他模块执行前对其进行度量。
- **Regular Operation:** 在成功启动后的正常操作, 跟没有启动 TXT 时的操作一样。不过 TXT 相关资源是会通过操作系统和 VMM 得到保护的。
- **MLE Shut Down:** 在操作系统/VMM 对平台执行关机或重启操作前需要执行一些特定的步骤来退出安全环境, 然后才能关机或重启。

TXT/TBoot 可信启动已经在 2.6.33 之后的 Linux Kernel 和 3.4 之后的 Xen 中得到支持。各大 Linux 发行版的最新版本中也直接提供了 TBoot 安装包。

8.3.3 可信认证与 OpenAttestation 项目

简单地说, 可信认证过程提供了将可信启动后的系统完整性度量值与白名单进行比较的能力。可信认证可以识别系统的可信状态, 为满足可视和可审查的需求提供帮助。图 8-10 所示为基于 TPM 的可信认证标准流程。

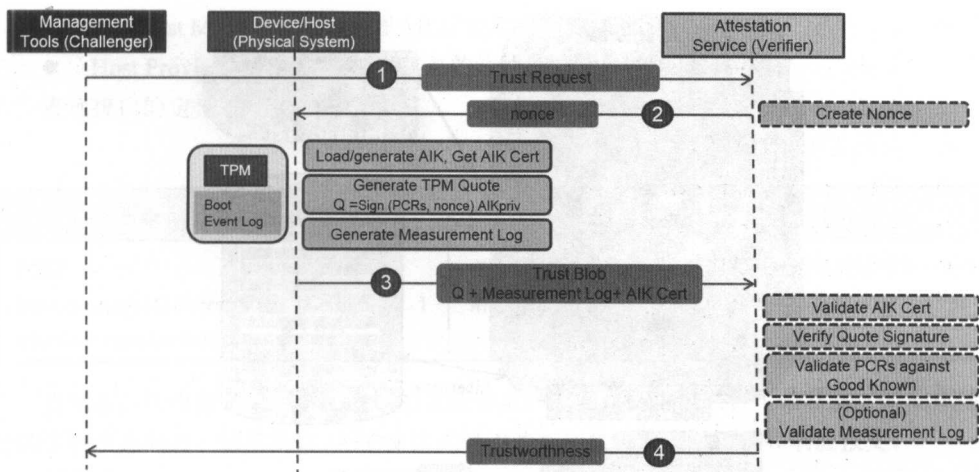


图 8-10 可信认证流程

从图 8-10 中不难看出,可信认证定义了 3 个主要实体:管理工具、认证服务和物理主机。认证的流程如下:

- 1) 管理工具向认证服务发起认证请求。
- 2) 认证服务生成一个随机数,发送给指定物理主机并请求完整性报告。
- 3) 物理主机。
 - 通过本地 TPM 模块生成/装载认证专用密钥 (AIK)。
 - 取得由 CA 颁发的认证专用证书 (AIC)。
 - 调用 TPM 模块的 Quote 命令生成以 AIK 为签名密钥的指定 PCR 集合的签名信息。
 - 生成度量日志。
 - 然后将由签名信息、度量日志和 AIC 组成的完整性报告发回到认证服务。
- 4) 认证服务。
 - 验证 AIC 的合法性。
 - 验证 Quote 签名信息。
 - 通过白名单验证 PCR 集合的可信状态。
 - 选择性验证度量日志的内容。
 - 将认证结果发回管理工具:指定物理主机可信/不可信。

由 Intel 于 2012 年创建的 OpenAttestation (OAT) 开源项目,使用 TCG 定义的远程认证协议,实现了标准的基于 TPM 的可信认证流程,为云计算及企业数据中心管理工具提供管理主机完整性验证的 SDK。至今发布过 1.0, 1.5, 1.6, 1.7, 2.0, 2.1, 2.2 共 7 个版本,最新的 2.2 版为 2014 年年底发布。OAT 体系结构如图 8-11 所示。

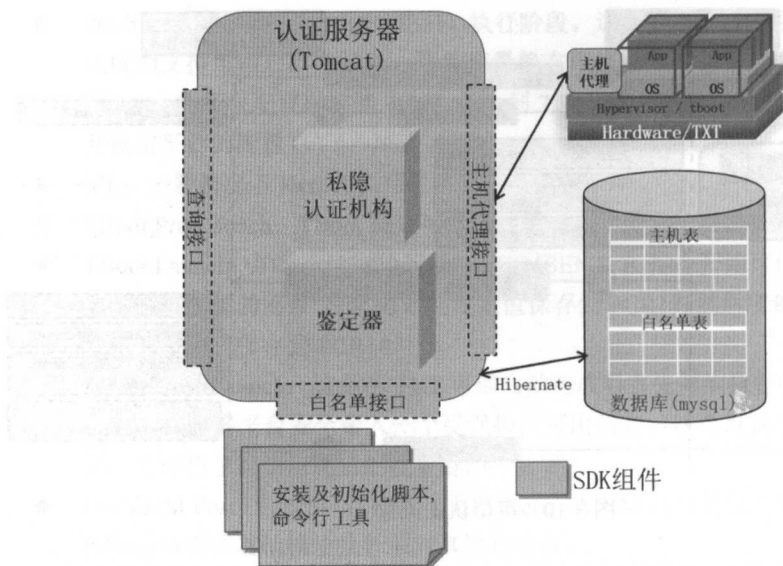


图 8-11 OAT 体系结构

OAT 主要包括两大组件：认证服务器和主机代理。

- 认证服务器有 5 个主要模块：隐私认证机构（PCA）、鉴定器（Appraiser）、查询接口（Query API）、白名单接口（Whitelist API）和主机代理接口（Host Agent API）。这 5 个模块均为 Java 代码，其中查询接口和白名单接口基于 RESTful 接口标准实现，主机代理接口为 Web Services 实现。
- 主机代理则包括代理服务、TPM 模块 Java 工具、NIARL TPM 管理工具等模块。代理服务通过 TPM 模块 Java 工具从命令行调用基于 TrouSerS 的 NIARL TPM 管理工具访问 TPM 来取得相关密钥、生成证书及完整性报告。

白名单接口的定义如表 8-1 所示。

表 8-1 白名单接口定义

接口类型	描 述
OEM Provisioning	Add/Update/Delete/Retrieve OEM
OS Provisioning	Add/Update/Delete/Retrieve OS
MLE Provisioning	Add/Update/Delete/Retrieve MLE
WhiteList Mgt	Add/Update/Delete PCR whitelist entry for a specified MLE
Host Provisioning	Register/update/Delete/Retrieve host

- OEM Provisioning: 管理对于 OEM 实体的增删改查。
- OS Provisioning: 管理对于 OS 实体的增删改查。
- MLE Provisioning: 管理对于 MLE 实体的增删改查。

- Whitelist Management: 管理 MLE 实体中 PCR 白名单的增删改。
- Host Provisioning: 管理主机信息的注册、注销和改查操作。

查询接口的定义如表 8-2 所示。

表 8-2 查询接口的定义

命令	输入参数	输出参数	注解
POST https://server:8181/AttestationService/resources/PollHosts	签权字段 {主机名, ...}	主机可信状态数据, 指定 PCR 的值	同步查询并等待验证服务获得主机可信状态以及相应 PCR 的值

查询接口接收指定主机名字列表的认证请求, 返回包括主机名、可信状态、时间戳等信息的主机状态列表。这里是以 JASON 格式描述的查询接口的一对请求和响应报文。

请求报文:

```
POST AttestationService/resources/PollHosts
Host: Attestation.ras.com:8181
Context-Type: application/json
Accept: application/json
Auth_blob: authenticationBlob
{
  "hosts": [host1.compute.com]
}
```

响应报文:

```
HTTP/1.1 200 OK
Server: BaseHTTP/0.3 Python/2.7.1+
Date: Wed, 24 Aug 2011 03:19:56 CST
Context-Type: application/json
{
  "hosts": [{"host_name": "host1.compute.com",
    "trust_lvl": "trusted",
    "vtime": "2011-08-24T03:19:56.376+08:00"}]
}
```

最后让我们通过图 8-12 来了解一下 OAT 的部署和使用步骤。

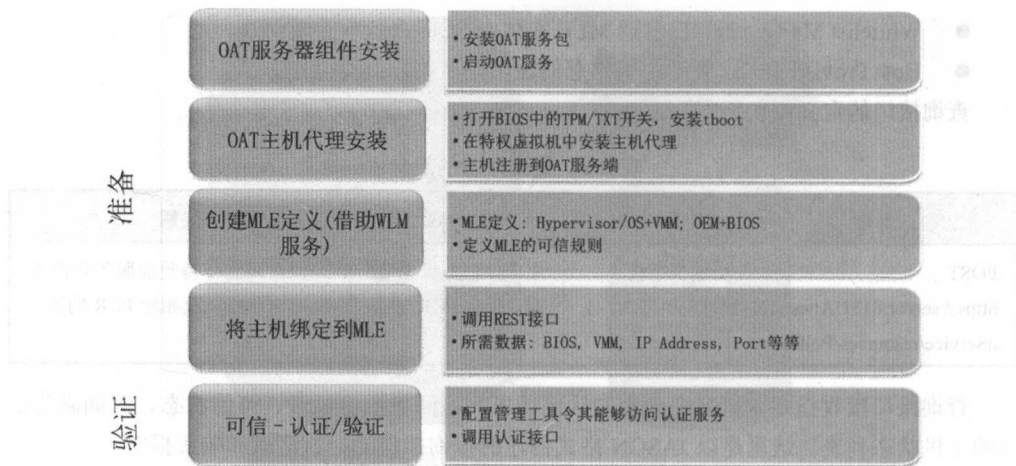


图 8-12 OAT 部署及使用步骤

OAT 的部署分为安装和配置两部分, 分别安装好 OAT 服务器组件和主机代理后, 借助白名单接口创建好可信的 MLE 定义, 并将主机绑定到相应的 MLE 定义上, OAT 的部署就完成了。接下来只要配置好云环境/数据中心管理工具, 就可以通过管理工具调用查询接口来对指定的物理主机进行远程认证了。

8.3.4 TrustedFilter

TrustedFilter 的实现位于 nova/scheduler/filters/trusted_filter.py。TrustedFilter 使用了一个新的配置选项组 trusted_computing, 并在 trusted_computing 下注册了 7 个新的配置选项, 用来配置连接 OAT 服务:

```
trust_group = cfg.OptGroup(name="trusted_computing",
                           title="Trust parameters",
                           help="")
Configuration options for enabling Trusted Platform Module.
"""

trusted_opts = [
    # OAT 服务器域名或 IP
    cfg.StrOpt("attestation_server",
               help='The host to use as the attestation server...'),
    # OAT 服务的 SSL 证书
    cfg.StrOpt("attestation_server_ca_file",
               help="The absolute path to the certificate to use for authentication when connecting to the attestation server..."),
    # OAT 服务的 https 端口
```

```

        cfg.StrOpt("attestation_port",
            default=8443,
            help=""The port to use when connecting to the attestation
server... """),
        # OAT 服务的 web API URL
        cfg.StrOpt("attestation_api_url",
            default="/OpenAttestationWebServices/V1.0",
            help=""
The URL on the attestation server to use... """),
        # OAT 服务的 authorization blob
        cfg.StrOpt("attestation_auth_blob",
            secret=True,,
            help=""
Attestation servers require a specific blob that is used to authenticate... """),
        # OAT 服务的状态缓存有效时间
        cfg.IntOpt("attestation_auth_timeout",
            default=60,
            help=""This value controls how long a successful attestation
is cached... """),
        # 禁止 OAT 服务 SSL 证书验证
        cfg.BoolOpt("attestation_insecure_ssl",
            default=False,
            help=""When set to True, the SSL certificate verification
is skipped for the attestation service... """)
    ]

```

每当 TrustedFilter 实例被创建时，会有一个 ComputeAttestation 实例被创建，进而有一个 ComputeAttestationCache 实例被创建。在 ComputeAttestationCache 实例的创建过程中，会通过 admin 上下文拿到系统中所有计算节点的列表，然后在 ComputeAttestationCache._init_cache_entry()中初始化每个节点可信状态缓存。

当 TrustedFilter.host_passes() 被调用并且 filter_properties.instance_type.extra_specs.'trust:trusted_host'不为空时，TrustedFilter.compute_attestation.is_trusted()调用 ComputeAttestation.caches.get_host_attestation()发现某个主机的缓存值失效，于是调用 ComputeAttestationCache._update_cache() 批量查询并更新所有主机的缓存值。接下来的一段时间(<attestation_auth_timeout>秒)里发生的主机可信状态查询都会命中缓存，从而避免连接 OAT 服务查询所带来的性能损失。

ComuputeAttestationCache.get_host_attestation()只要发现任何一个主机可信状态的缓存失效就会查询并更新所有主机的状态。这样设计的初衷是，充分利用在 OAT 查询命令中出现多主机查询时会进行并行处理的特性，尽量将一次虚拟机创建请求可能产生的对所有主机状态的串行化查询并行化，从而改善使用 TrustedFilter 时的创建效率。

所有与 OAT 服务通信的操作都封装在 AttestationService 类中来处理对 OAT 服务查询接口

的调用和返回值处理。

8.3.5 部署

正式开始配置可信计算池之前，需要：

- 部署好基本的 OpenStack 运行环境，准备至少两台主机（计算节点）。
- 在单独的服务器（物理机或虚拟机）上安装好 OAT 服务，并将所需主机注册为可信主机。

1. 配置

获得 OAT 服务端 SSL 证书：

```
$ openssl s_client -connect <OAT_APPRAISER_HOSTNAME>:8181 | tee /etc/nova/certfile.cer
```

验证计算节点已经注册为可信主机：

```
$ curl --noproxy <OAT_APPRAISER_HOSTNAME> -v --cacert ./certfile.cer -H "Content-Type: application/json" -X POST -d '{"hosts":["<OAT_CLIENT_HOSTNAME>"]}' https://<OAT_APPRAISER_HOSTNAME>:8181/AttestationService/resources/PollHosts
```

修改 nova 调度器所在系统的/etc/nova/nova.conf：

```
# 在 DEFAULT 一节中添加以下内容来打开调度器对可信计算池的支持
[DEFAULT]
compute_scheduler_driver=nova.scheduler.filter_scheduler.FilterScheduler
scheduler_available_filters=nova.scheduler.filters.all_filters
scheduler_default_filters=AvailabilityZoneFilter,RamFilter,ComputeFilter,TrustedFilter

# 在 trusted_computing 一节中添加以下内容来给定认证服务的连接信息
[trusted_computing]
attestation_server=<OAT_APPRAISER_HOSTNAME>
attestation_port=8181
attestation_api_url=/AttestationService/resources
attestation_server_ca_file=/etc/nova/certfile.cer
attestation_attestation_auth_blob=oatoat
attestation_auth_timeout=60
attestation_insecure_ssl=False
```

使用 nova flavor-key set 命令将一个或多个 flavor 设置为可信 flavor：

```
$ nova flavor-key ml.trusted set trust:trusted_host=trusted
```

2. 使用

通过命令行指定一个可信 flavor 创建实例来请求将其运行在可信主机上：

```
$ nova boot --flavor m1.trusted --image <image_id> --key_name <keypair>  
myinstance
```

或者通过 Horizon Dashboard 来创建可信实例：

- 找到 Project→Images and Snapshots，选择需要运行的镜像，单击 launch。
- 在接下来的屏幕上选择 m1.trusted 作为镜像的 flavor，然后新的实例只会在被 OAT 认证为可信平台的主机上被创建出来。
- 在 Horizon Dashboard 界面上进入 Admin 属性页，单击 Instances 来显示运行中的实例。
- 验证所创建的实例已经在可信服务器上运行起来。

计量与监控

计量与监控是云运营的一个重要环节。

OpenStack Newton 版本中由 Telemetry 项目来提供计量与监控的功能。Telemetry 的目标是对各种组成云的虚拟和物理资源，收集各类使用数据并保存，以便对这些数据进行后续分析，并在相关条件满足时，可以触发对应的动作和报警。

Telemetry 项目中有许多子项目，其中包括以下几个。

- Aodh: 根据已保存的使用数据进行报警。
- Ceilometer: 提供一个获取和保存各类使用数据的统一框架。
- Gnocchi: 用来保存基于时间序的数据，并对其提供索引服务。
- Panko: 用来保存事件 Event 信息。

其中，Ceilometer 处于核心位置，OpenStack Telemetry 的源头即是 Ceilometer 项目。它开始于 2012 年，在 2013 年 OpenStack Grizzly 版本中从孵化状态毕业，成为 OpenStack 的集成项目（Integrated Project），并从 Havana 版本开始，被正式包含在 OpenStack 发布版本中。Ceilometer 的目标是为 OpenStack 环境提供一个获取和保存各种测量值（Measurement Metrics）的统一框架。

Ceilometer 的最初目的只是为了获取保存计量信息来支持对用户收费。后来随着项目的发展，社区发现很多 OpenStack 项目都需要获取保存很多种不同的测量值，所以 Ceilometer 项目又增加了第二个目标，成为 OpenStack 系统里一个标准的获取保存测量值的框架。后来，由于 Heat 项目的需求，Ceilometer 又增加了利用已保存的测量值进行报警的功能。

Ceilometer 之外，其他项目的功能都是由 Ceilometer 中剥离出来，或者为了改进 Ceilometer 中的不足而后续开发的。

为了满足某些客户需要更加灵活的有选择的部署 Ceilometer 各部分功能的目标，在 OpenStack Liberty 版本中，报警功能从 Ceilometer 中剥离，产生了一个新项目 Aodh。在 OpenStack Newton 版本中，事件 Event 信息的保存也被剥离，产生了一个新项目 Panko。

同时，为了解决 Ceilometer 原先在数据存储端的一些设计上的性能弊病，Telemetry 社区开发了一个新项目 Gnocchi，用来保存基于时间序的数据，作为 Ceilometer 的数据保存后端。

这些从 Ceilometer 中分离和衍生出来的子项目与 Ceilometer 一起统一构成了 OpenStack Telemetry 项目。所有这些项目目前都属于 OpenStack Big Tent。

本章主要介绍 Ceilometer，同时也对其他项目做一些简单的介绍。在本章内容完成时，OpenStack Telemetry 社区正在讨论一些架构性的较大改动，比如取消 Ceilometer API、移除

Ceilometer collector 等。这些架构改动由于还处在可行性讨论阶段，这里并不做反映。

9.1 Ceilometer

9.1.1 体系结构

Ceilometer 整体采用了高度可扩展性的设计思想，开发者可以很容易地开发扩展插件来扩展其功能。Ceilometer 的整体架构如图 9-1 所示。

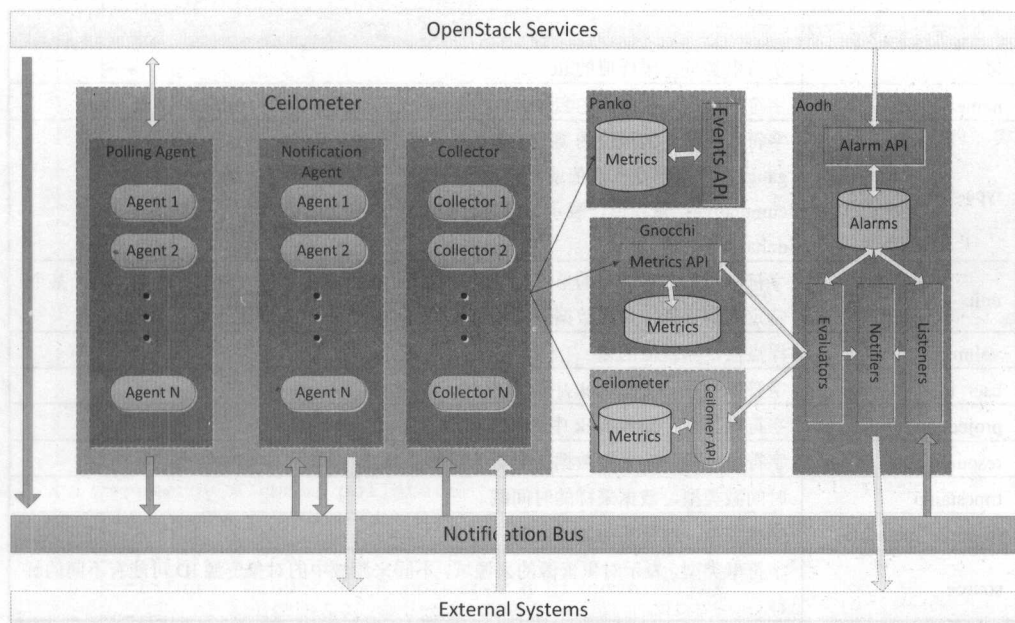


图 9-1 Ceilometer 整体架构

Ceilometer 可以通过以下 3 种方式获取测量值数据：

- 第三方的数据发送者把数据以通知消息（Notification Message）的形式发送到消息总线（Notification Bus）上，Notification Agent 会获取这些通知事件，从中提取测量数据。
- Polling Agent 会根据配置，定期主动通过各种 API 或者其他通信协议去远端或者本地的不同服务实体中获取所需要的测量数据。
- 用户可以通过调用 RESTful API 直接把利用其他方式获取的任意测量数据送达给 Ceilometer。

Ceilometer 通过以上 3 种方法获取到测量值数据后，会把它转化为符合某种标准格式的数

据采样（Sample）通过内部总线发送给 Notification agent。然后 Notification Agent 根据用户定义的 Pipeline 来对数据采样进行转换（Transform）和发布（Publish）。如果根据 Pipeline 的定义，这个数据采样（Sample）最后被发布给 Collector 的话，Collector 会把这个数据采样保存在数据库中。

用户可以通过 Ceilometer API 对保存在数据库中的测量数据进行条件查询、数据聚合（Data Aggregation）等操作。

Ceilometer 内部采用如表 9-1 所示的统一格式来表示测量的采样数据 Sample。

表 9-1 数据采样（Sample）的字段格式

字 段	说 明
id	字符串类型，采样值的 id
name	字符串类型，测量值的名称
type	字符串类型，测量值的类型，支持以下 3 个值 gauge: 用来测量离散值或者波动值 cumulative: 累积值，表示值随着时间而增加 delta: 变化量
unit	字符串类型，测量值的单位，一般建议采用国际标准单位（SI）。注意，对于某个特定的测量值，采样数据中的单位要保持一致，不能有变化
volume	浮点值，测量值的量
user_id	字符串类型，openstack 中的用户 ID
project_id	字符串类型，openstack 中的 tenant ID
resource_id	字符串类型，此采样数据所测量的对象资源 ID
timestamp	时间戳类型，数据采样的时间戳
resource_metadata	字典类型，对象资源的元数据
source	字符串类型，表示对象资源的来源域，不同来源域中的对象资源 ID 可能有不同的解释

Ceilometer 源代码的部分目录结构如下：

```
.
| requirements.txt - 运行时所需的第三方 ython 库
| setup.cfg - setuptools 配置文件
| setup.py
| test-requirements.txt - 测试时所需的第三方 python 库
| tox.ini - 测试环境配置
|-- ceilometer
|   | collector.py - collector 的实现
|   | coordination.py - 多个 agent 之间的同步机制
|   | declarative.py - JSONPATH 格式的 yaml 文件解析
|   | exchange_control.py
|   | gnocchi_client.py - gnocchi 客户端封装
```

- | | i18n.py - 国际化实现
- | | keystone_client.py - keystone 客户端封装
- | | messaging.py - oslo.messaging 封装
- | | middleware.py
- | | neutron_client.py - neutron 客户端封装
- | | notification.py - notification agent 的实现
- | | nova_client.py - nova 客户端封装
- | | opts.py - 配置选项汇总
- | | pipeline.py - pipeline 的实现
- | | sample.py - 测量取样的定义
- | | service.py - service 初始化
- | | service_base.py - polling agent 和 notification agent 的 service 基类
- | └agent
 - | | manager.py - polling agent 的实现
 - | | plugin_base.py - pollster/notification listener/discover 的基类
 - | └discovery - 通用 discover 实现
- | └api - ceilometer api 的实现
- | └cmd - ceilometer 运行程序的外层分装
- | └compute
 - | | discovery.py - computer pollster 的默认 discover
 - | | util.py
 - | └notifications - nova notification listener
 - | └pollsters - compute polling agent pollsters
 - | └virt - compute polling agent inspectors
- | └dispatcher - collector 的各类 dispatcher 实现
- | └energy - kwapi pollsters
- | └event - 生成 Event 事件数据
- | └hardware - snmp pollster 和 discover 的实现
- | └image - glance 相关的 pollster 和 discover 的实现
- | └ipmi - ipmi polling agent 相关 pollster
- | └meter - 通用的 meter notification listener 实现
- | └network
 - | | floatingip.py - neutron pollster
 - | | notifications.py - neutron notification listeners
 - | └services - fwaas/lbaas/vpnaas pollsters
 - | └statistics - SDN 相关的 pollster
- | └objectstore - swift pollsters/listeners
- | └publisher - 各类 publisher 的实现
- | └storage - 测量值数据库的 Model/Driver
- | └telemetry - polling agent/API server 和 notification agent 间通讯
- | └tests - 测试用例
- | └transformer - pipeline transformers
- | └devstack - 用于使用 devstack 安装
- | └doc - 开发者文档目录

接下来将详细介绍 Ceilometer 项目框架中的各个重要组成部分。

9.1.2 Pipeline

在 Ceilometer 中，适用于不同应用场景下的测量数据，采样频率可能有不同的要求。例如，对于用来计费的数据，采样的频率比较低，典型值为 5~30min；而对于用来监控的数据，采样频率就会高很多，一般会达到 1~10s。

另外，对于测量数据的发布方式，不同的应用场景也会提出不同的要求。对于计费的数据，就要求数据采样值不能丢失，并且要保证数据的不可否认性（non-repudiation）；而对于用来监控的数据，这方面就没有这么严格的要求。

为此，Ceilometer 中引入了 Pipeline 的概念来解决采样频率和发布方式的问题。Pipeline 概念的引入，使得管理员可以很容易地定义某一个测量数据的采样频率和发布方式。

Ceilometer 中，同时允许有多个 Pipeline，每个 Pipeline 都是由源（source）和目标（sink）两部分组成的。源中定义了需要测量哪些数据，数据的采样频率，在哪些 endpoint 上进行数据采样，以及这些数据的目标（sink）。目标中定义了获得的数据要经过哪些 Transformer 进行数据转换，并且最终交由哪些 Publisher 进行数据发布。图 9-2 所示为 Pipeline 的一个数据的转换发布流程。

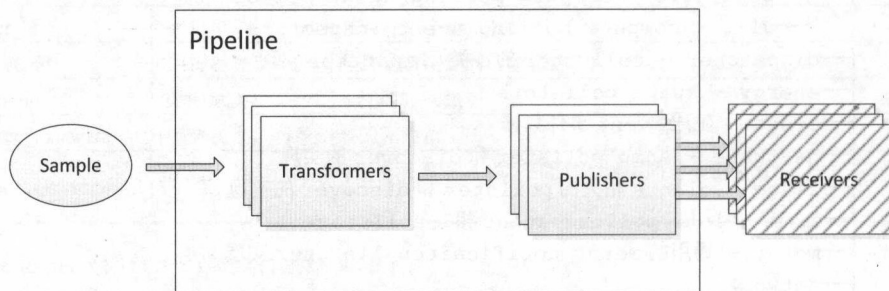


图 9-2 Ceilometer Pipeline 流程

转换器（Transformers）可以针对一个或者多个同一种类的数据采样值进行各种不同的操作，例如改变单位、聚合计算等，最终转换成一个或者多个其他不同种类的测量数据。图 9-3 所示是一个将多个 CPU 时间的测量采样值通过一个转换器转换为 CPU 使用率的测量采样值的示例。

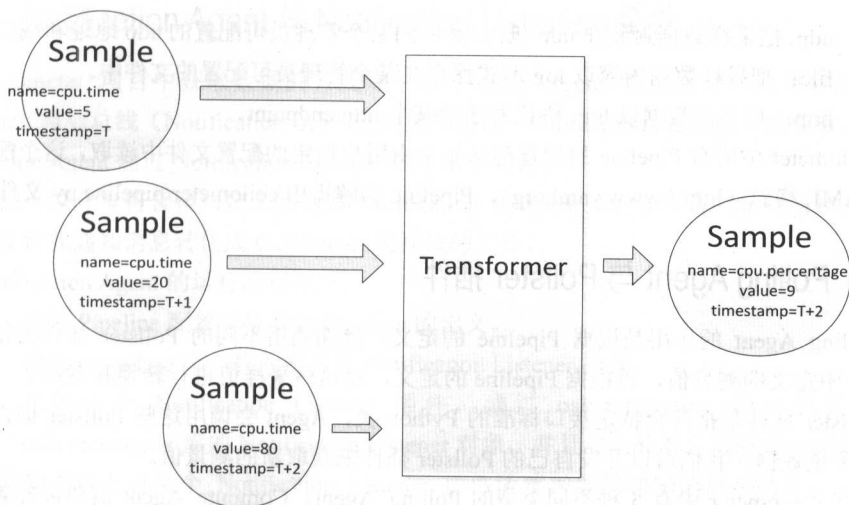


图 9-3 Transformer 示例

通过转换器转化后的数据，最终会交由 Pipeline 定义中的 Publisher 进行数据发布。图 9-4 所示的 Pipeline 中定义了 3 个不同的 Publisher，分别采用 message bus 上的 notification、kafka 消息和 udp 数据包 3 种不同的方式同时将一个数据采样值发布给不同的数据接收者。

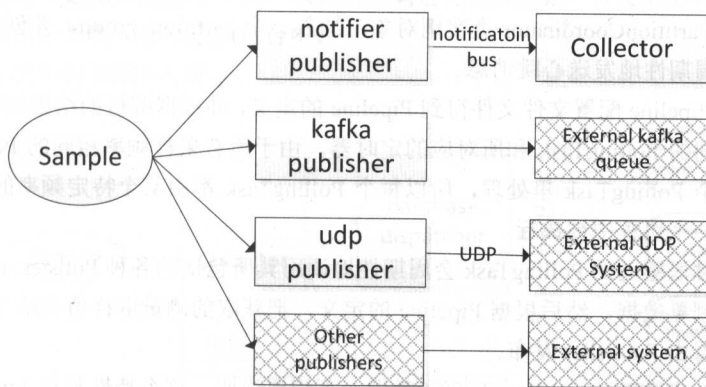


图 9-4 Publisher 示例

目前已有多种 Publisher，具体如下。

- **notifier**: 向通知总线上发送 info 级别的通知消息，采样数据（Sample）内容作为通知消息的数据载荷（payload）。
- **kafka**: 向 kafka 消息队列发送。
- **direct**: 直接调用 collector dispatcher 保存到数据后台，绕开 collector。

- **udp**: 把采样数据封装在 **udp** 包内, 然后向某个管理员可配置的 **udp** 地址和端口发送。
- **file**: 把采样数据内容以 **log** 形式保存在某个管理员可配置的文件中。
- **http**: 把采样数据以 **http** 协议发送给某个 **http endpoint**。

Ceilometer 中所有 Pipeline 的配置都从一个由用户指定的配置文件中读取, 这个配置文件采用 **YAML** 格式 (<http://www.yaml.org>), Pipeline 的解析由 `ceilometer/pipeline.py` 文件负责。

9.1.3 Polling Agent 与 Pollster 插件

Polling Agent 的作用是根据 Pipeline 的定义, 定期调用不同的 Pollster 插件去轮询获得 Pipeline 中定义的测量值, 再根据 Pipeline 的定义, 对这些采样值进行转换和发布。

Pollster 插件是指符合特定接口标准的 Python 类, Agent 会调用这些 Pollster 插件来获得不同的测量数据。我们可以开发自己的 Pollster 插件来获取新的测量值。

目前 Ceilometer 中有 3 种不同类型的 Polling Agent。Compute Agent 需要被部署在运行 Nova Compute 服务的计算节点上, 主要是用来和 Hypervisor 进行通信的, 轮询获取 Hypervisor 相关的测量值。Central Agent 可以被部署在任何结点上, 它用来和远程的各种不同的实体和服务进行通信, 获取不同的测量值。在 Juno 版本中, 又加入一个新的 IPMI Agent, 它需要部署在支持 IPMI 的节点上, 用来获取本机 IPMI 的相关测量值。

各种不同 Polling Agent 的运行流程基本比较类似:

- 调用 `stevedore` 库, 获取属于本 Agent 的所有 Pollster 插件。
- 创建 `PartitionCoordinator` 类实例对象, 加入一个 `partition group`, 并创建一个定时器用以周期性地发送心跳消息。
- 解析 Pipeline 配置文件文件得到 Pipeline 的定义, 并根据解析的结果创建一个或者多个不同的 `PollingTask` 和所对应的定时器。由于所有采样频率相同的 Pipeline 都会在同一 `PollingTask` 里处理, 所以每个 `PollingTask` 都由某个特定频率的定时器驱动, 在某一个协程中被执行。
- 由定时器驱动的 `PollingTask` 会周期性地调用其所包括的各种 Pollster, 由这些 Pollster 获取测量数据, 然后根据 Pipeline 的定义, 把获取的测量取样值交给 `Transformer` 转换后再由 `Publisher` 发布。

这里对第二步的 `PartitionCoordinator` 类做一下简要说明。这个特性是在 Juno 版本中加入的, 主要目的是消除 Juno 以前的版本中 polling Agent 只能部署一个实例所导致的两大缺点:

- 1) 当 Polling Agent 需要轮询许多测量信息时, 效率比较低下。
- 2) Polling Agent 成了所谓的 SPOF (Single Point of Failure)。PartitionCoordinator 类会利用 `tooz` 项目 (<https://github.com/stackforge/tooz>) 所提供的功能和一致性哈希算法, 允许同时部署多份 Central Agent 实例, 并且这些不同的 Central Agent 实例互相之间不会重复获取相同的测量数据。

9.1.4 Notification Agent 与 Notification Listeners 插件

Ceilometer 项目中获取数据的方式除了用 Polling Agent 轮询之外，还可以通过侦听 OpenStack 通知总线（Notification Bus）上的通知消息（notification message）来获取。这是由 Notification Agent 通过 Notification Listener 插件来实现的。

我们可以开发符合接口定义的新的 Notification Listener 插件来实现对新的通知消息的侦听，以及将此通知消息转化成 Ceilometer 采样值的工作。

Notification Agent 的运行流程如下：

- 解析 Pipeline 配置文件得到 Pipeline 的定义。
- 调用 stevedore 库，载入所有的 Notification Listener 插件。
- 对每一个 Notification Listener 插件，通过 oslo.messaging 库构造其对应的 oslo.messaging 库的 Notification Listener 对象，并且启动此对象监听通知消息。

当通知总线上有某个 Notification Listener 插件所感兴趣的通知消息到达时，所对应的 Notification Listener 插件就会被 Notification Agent 所调用，根据此通知消息构造出采样值 Sample 的实例对象，然后根据 Pipeline 中的定义将此采样值转换和发布出去。

9.1.5 Collector 与 Dispatcher 插件

当 Ceilometer 获得的测量采样数据通过 Pipeline 发布（Publisher）出去后，需要有一个数据接收者获得这些数据并且保存下来，以便对数据进行后续的进一步处理。Collector 的作用就是把所接收到的数据保存在数据后台中。

Collector 的架构如图 9-5 所示。

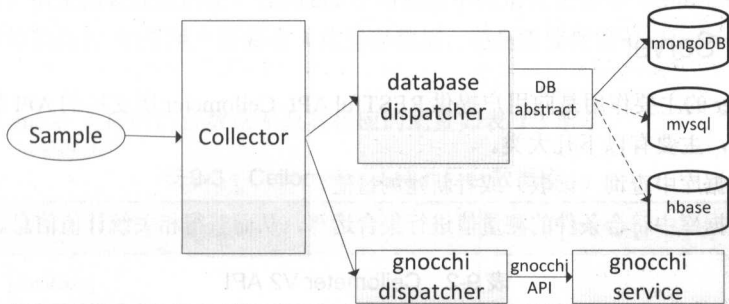


图 9-5 Collector 架构

对于每个 Collector，管理员可以为其配置一个或者多个 Dispatcher。对于每一个所收到的采样数据，Collector 会调用所有的 Dispatcher，由这些 Dispatcher 来决定如何处理数据。目前已有 4 个 Dispatcher。

- database dispatcher：把采样数据或者 Event 事件保存在后台数据库中。

- file dispatcher: 把采样数据或者 Event 事件以 log 的形式保存在文件中。
- gnocchi dispatcher: 把采样数据或者 Event 事件保存在 Gnocchi 中
- http dispatcher: 把采样数据或者 Event 事件以 http 封装发送给某个 http endpoint。

我们也可以通过实现 `ceilometer.dispatcher.MeterDispatcherBase` 类或者 `ceilometer.dispatcher.EventDispatcherBase` 来开发符合自己要求的 Dispatcher, 具体可以参见 `ceilometer/dispatcher/__init__.py` 文件中的定义。

9.1.6 Storage/DB

为了支持多种不同的后台数据库, Ceilometer 引入了数据库抽象层。通过数据库抽象层, 上层可以通过统一的接口向后台不同的数据库写入要保存的数据, 同时也可以通过统一的接口从后台不同的数据库获得以一致的格式所表示的数据。

目前, Ceilometer 支持 4 种不同的数据库后台: MongoDB、MySQL、PostgreSQL 和 Hbase。需要注意的是, 对于不同的数据库后台, 所支持的功能和性能有所不同, 也可以根据自己的需要进行选择。

和 OpenStack 其他项目相比, Ceilometer 所需要保存的数据量会大很多。所以虽然 Ceilometer 的默认后台数据库是 MySQL, 但是历史上一般建议在实际生产环境中采用 MongoDB 为后台数据库。MongoDB 作为非关系型数据库, 在保存处理大数据量时相比, MySQL 等关系型数据库有着更多、更好的特性。

Ceilometer 还支持开发者自己开发对其他数据库后台的支持。

注意, 由于性能等方面的原因, Telemetry 社区目前建议使用 Gnocchi 代替上面所说的数据库来保存采样数据。关于 Gnocchi, 具体参见第 9.3 节。

9.1.7 API Server

API Server 的主要作用是向用户提供 RESTful API。Ceilometer 所支持的 API 版本号是 V2。如表 9-2 所示, 主要有以下几大类。

- 从数据库中查询 (读取) 或者新建测量值。
- 对数据库中符合条件的测量值进行集合运算, 从而获得相关统计值信息。

表 9-2 Ceilometer V2 API

API URL	说 明
GET /v2/resources	获得符合查询条件的所有对象资源
GET /v2/resources/(resource_id)	获得某个特定对象资源
GET /v2/meters	获得符合查询条件的所有测量值
POST /v2/query/samples	获得符合用户自定义复杂查询的所有采样值
GET /v2/meters/(meter_name)	获得符合查询条件的某个特定测量值的所有采样值

续表

API URL	说 明
POST /v2/meters/(meter_name)	记录某个特定测量值的一个采样值
GET /v2/meters/(meter_name)/statistics	获得符合查询条件的某个特定测量值的统计信息
GET /v2/capabilities	查询此 API 版本所支持的功能

其中，集合运算的相关统计值除了一般的最大值、最小值、平均值、累加总和、总数之外，如果用户采用 MongoDB/MySQL/PostgreSQL 作为后台数据库，还支持求标准方差和求基数（cardinality）两个操作。

API Server 最外层采用 PasteDeploy 库来载入 Ceilometer 中的 WSGI 应用和其他 WSGI 中间件。Ceilometer API 目前默认使用中间件 keystonemiddleware，主要用来和 keystone 进行通信、对用户进行身份验证。

Ceilometer 的 WSGI 应用采用了基于 Pecan 的框架来构造 RESTful API 的实现。Pecan 是 Python 的一个轻量级 Web 框架的实现（具体参见 <http://pecan.readthedocs.org>）。此应用的具体实现代码可以参见 ceilometer/api/controllers/v2 目录。

9.1.8 部署与使用

1. 选择数据保存后台

Ceilometer 目前支持同时把数据保存到多个不同的数据后台。用户首先需要决定的是把数据保存到数据库后台还是 Gnocchi。对于基于时间序列的大数据量数据，社区建议把这类数据保存到 Gnocchi。Gnocchi 相比 Ceilometer 自身提供的数据库，有着更快的查询速度和更小的存储空间占用。但是需要注意的是，Gnocchi 中的数据不能永久化保存（Gnocchi 会自动删除超出时间窗口的数据），如果用户需要永久化保存数据，还是需要使用 Ceilometer 目前自身的数据库。

目前 Ceilometer 支持的后台数据库及其相应的配置如表 9-3 所示。

表 9-3 Ceilometer 支持的后台数据库

数据库	配置项示例	说 明
MongoDB	[database] metering_connection = mongodb://user:pass@host:27017/ceilometer	需要 MongoDB 2.4 及以后的版本
SqlAlchemy 所支持的关 系型数据库	[database] metering_connection= mysql://user:pass@host/ceilometer?charset=utf8	需要运行 ceilometer-dbsync 来创建数据库

续表

数据库	配置项示例	说 明
HBase	[database] metering_connection = hbase://hbase-thrift-host:9090	Ceilometer 使用 HappyBase 库通过 HBase Thrift 接口和后台 Hbase 数据库进行通信

Ceilometer 中不同的数据库后台所支持的功能也有所不同，如表 9-4 所示。

表 9-4 Ceilometer 支持的后台数据库功能对比

功 能		MongoDB	SqlAlchemy 所支持的 关系型数据库	HBase
测量值	基本查询	✓	✓	✓
	基于 metadata 的查询	✓	✓	✓
测量取样	基本查询	✓	✓	✓
	基于 metadata 的查询	✓	✓	✓
	用户自定义复杂查询	✓	✓	✗
集合统计	基本查询	✓	✓	✓
	基于 metadata 的查询	✓	✓	✓
	支持 groupby 操作	✓	✓	✗
	支持基本统计操作： max/min/avg/count/sum	✓	✓	✓
	支持标准方差统计	✓	✓	✗
	支持基数（cardinality）操作	✓	✓	✗

2. 安装

和其他 OpenStack 项目类似，安装 Ceilometer 有以下几种不同的方式：

（1）从 Linux 发行版的安装包安装

通过包管理工具安装是最简单的方式。由于不同的 Linux 发行版有着不同的包管理工具，具体请参见 <http://docs.openstack.org/> 网站上的安装指南（Installation guide）。

（2）使用 Devstack 安装

大部分的开发者都会使用 Devstack 来安装配置开发环境。相比与使用安装包，这种方式可以安装最新的 Ceilometer 代码。用户可以在 Devstack 的 localrc 配置文件中设置如下选项来安装 Ceilometer（具体可以参见 <http://docs.openstack.org/developer/ceilometer/install/development.html>）：

```
[[local|localrc]]
enable_plugin ceilometer https://git.openstack.org/openstack/ceilometer.git
```

#默认 ceilometer 会部署除 ipmi-agent 之外的所有服务，如果要部署 ipmi-agent 的话，可以打

开下面的语句

```
# enable_service ceilometer-aipmi
```

(3) 手动安装

安装数据后台：如果选择 Gnocchi 作为数据后台，具体步骤如下：

- 1) 安装 Gnocchi，具体参见第 9.3 节。
- 2) 初始化 Gnocchi。

```
$ gnocchi-upgrade --create-legacy-resource-types
```

3) 在文件 `/etc/ceilometer/ceilometer.conf` 中，为 collector service 配置如下相关选项：

```
[DEFAULT]
meter_dispatchers = gnocchi
event_dispatchers = gnocchi

[dispatcher_gnocchi]
filter_service_activity = False # Enable if using swift backend
filter_project = <project name associated with gnocchi user> # if using swift
backend *

[service_credentials]
auth_url = <auth_url>:5000
region_name = RegionOne
password = password
username = ceilometer
project_name = service
project_domain_id = default
user_domain_id = default
auth_type = password
```

4) 复制 `gnocchi_resources.yaml` 到 `ceilometer` 配置目录 (`/etc/ceilometer/`)。

5) 打开如下 `Ceilometer` 的配置选项，可以提升性能。

```
[cache]
backend_argument = redis_expiration_time:600
backend_argument = db:0
backend_argument = distributed_lock:True
backend_argument = url:redis://localhost:6379
backend = dogpile.cache.redis

[collector]
batch_size = 100
batch_timeout = 5
```

6) 如果有需要的话，可能需要重启 `ceilometer collector` 服务。

如果选择数据库作为后台，具体请参见本节最开始的内容。注意，如果选择了基于

SqlAlchemy 所支持的关系型数据库（比如 MySQL、PostgreSQL），需要运行以下命令来创建数据库表。

```
# 下面以 MySQL 为例
# 创建数据库
$ mysql -u<user> -p<pass> -h<mysql_host> -e "CREATE DATABASE ceilometer
CHARACTER SET utf8;"

# 在 ceilometer 配置文件中设置 database connection 选项为
# mysql://<user>:<pass>@<mysql_host>/ceilometer?charset=utf8
# 具体请参见表 3
$ vim /etc/ceilometer/ceilometer.conf

# 创建（升级）数据库表定义
$ ceilometer-dbsync
```

在 Keystone 中创建 ceilometer 用户，具体代码如下：

```
# 创建 ceilometer 用户
$ openstack user create --domain default --password-prompt ceilometer
User Password:
Repeat User Password:

+-----+-----+
| Field      | Value                                     |
+-----+-----+
| domain_id  | e0353a670a9e496da891347c589539e9 |
| enabled    | True                                |
| id         | c859c96f57bd4989a8ea1a0b1d8ff7cd |
| name       | ceilometer                           |
+-----+-----+

# 给 ceilometer 用户赋予 admin 角色
$ openstack role add --project service --user ceilometer admin
```

安装 notification agent，具体代码如下：

```
# 获取代码
$ git clone https://git.openstack.org/openstack/ceilometer.git
# 安装
$ cd ceilometer
$ sudo python setup.py install
# 产生配置文件
$ tox -egenconfig
# 复制配置文件
$ mkdir -p /etc/ceilometer
$ cp etc/ceilometer/*.yaml /etc/ceilometer
$ cp etc/ceilometer/*.json /etc/ceilometer
```



```

$ cp etc/ceilometer/*.conf /etc/ceilometer
$ etc/ceilometer/ceilometer.conf.sample /etc/ceilometer/ceilometer.conf
# 编辑配置文件/etc/ceilometer/ceilometer.conf, 修改如下相关选项
# oslo_messaging 相关
[oslo_messaging_notifications]
topics = notifications
[oslo_messaging_rabbit]
rabbit_userid = stackrabbit
rabbit_password = openstack1
rabbit_hosts = 10.0.2.15
# message 签名相关, 如果设为空, 表示禁止 message 签名功能, 注意对于所有 ceilometer
service, 都必须使用相同的 telemetry_secret
[DEFAULT]
telemetry_secret=<secret value>

```

安装 collector, 具体代码如下:

```

# 获取代码
$ git clone https://git.openstack.org/openstack/ceilometer.git
# 安装
$ cd ceilometer
$ sudo python setup.py install
# 产生配置文件
$ tox -egenconfig
# 复制配置文件
$ mkdir -p /etc/ceilometer
$ cp etc/ceilometer/*.yaml /etc/ceilometer
$ cp etc/ceilometer/*.json /etc/ceilometer
$ cp etc/ceilometer/*.conf /etc/ceilometer
$ etc/ceilometer/ceilometer.conf.sample /etc/ceilometer/ceilometer.conf
# 编辑配置文件/etc/ceilometer/ceilometer.conf, 修改如下相关项
# oslo_messaging 相关
[oslo_messaging_notifications]
topics = notifications
[oslo_messaging_rabbit]
rabbit_userid = stackrabbit
rabbit_password = openstack1
rabbit_hosts = 10.0.2.15
# message 签名相关, 如果设为空, 表示禁止 message 签名功能, 注意对于所有 ceilometer
service, 都必须使用相同的 telemetry_secret
[DEFAULT]
telemetry_secret=<secret value>

```

安装 polling agent, 具体代码如下:

```

# 获取代码

```



```

$ git clone https://git.openstack.org/openstack/ceilometer.git
# 安装
$ cd ceilometer
$ sudo python setup.py install
# 产生配置文件
$ tox -egenconfig
# 复制配置文件
$ mkdir -p /etc/ceilometer
$ cp etc/ceilometer/*.yaml /etc/ceilometer
$ cp etc/ceilometer/*.json /etc/ceilometer
$ cp etc/ceilometer/*.conf /etc/ceilometer
$ etc/ceilometer/ceilometer.conf.sample /etc/ceilometer/ceilometer.conf
$ cp -r etc/ceilometer/rootwrap.d /etc/ceilometer
# 编辑配置文件/etc/ceilometer/ceilometer.conf, 修改如下相关项
# oslo_messaging 相关
[oslo_messaging_notifications]
topics = notifications
[oslo_messaging_rabbit]
rabbit_userid = stackrabbit
rabbit_password = openstack1
rabbit_hosts = 10.0.2.15
# message 签名相关, 如果设为空, 表示禁止 message 签名功能, 注意对于所有 ceilometer
service, 都必须使用相同的 telemetry_secret
[DEFAULT]
telemetry_secret=<secret value>

```

安装 Ceilometer API, 具体代码如下:

```

# 获取代码
$ git clone https://git.openstack.org/openstack/ceilometer.git
# 安装
$ cd ceilometer
$ sudo python setup.py install
# 产生配置文件
$ tox -egenconfig
# 复制配置文件
$ mkdir -p /etc/ceilometer
$ cp etc/ceilometer/api-paste.ini /etc/ceilometer
$ cp etc/ceilometer/*.yaml /etc/ceilometer
$ cp etc/ceilometer/*.json /etc/ceilometer
$ cp etc/ceilometer/*.conf /etc/ceilometer
$ etc/ceilometer/ceilometer.conf.sample /etc/ceilometer/ceilometer.conf
# 编辑配置文件/etc/ceilometer/ceilometer.conf, 修改如下相关项
# oslo_messaging 相关
[oslo_messaging_notifications]
topics = notifications

```

```

[oslo_messaging_rabbit]
rabbit_userid = stackrabbit
rabbit_password = openstack1
rabbit_hosts = 10.0.2.15
# message 签名相关, 如果设为空, 表示禁止 message 签名功能, 注意对于所有 ceilometer
service, 都必须使用相同的 telemetry_secret
[DEFAULT]
telemetry_secret=<secret value>

# 为 Ceilometer 在 keystone 中新建一个服务
$ keystone service-create --name=ceilometer \
                           --type=metering \
                           --description="Ceilometer Service"

# 为 Ceilometer 建立 endpoint
# CEILOMETER_SERVICE 是上条命令中返回的 service_id
# SERVICE_HOST 是部署了 Ceilometer API server 的机器 ip 地址
# 默认端口号是 8777, 如果管理员在配置文件中设置了其他端口号, 这里需要做相应的调整
$ keystone endpoint-create --region RegionOne \
                           --service_id $CEILOMETER_SERVICE \
                           --publicurl "http://$SERVICE_HOST:8777/" \
                           --adminurl "http://$SERVICE_HOST:8777/" \
                           --internalurl http://$SERVICE_HOST:8777/

```

配置其他 OpenStack 项目: 如果需要 Ceilometer Notification Agent 能够侦听其他 OpenStack 项目的通知消息, 首先需要配置其他项目的 Notification Bus。

1) Glance, 编辑 glance.conf 中加入以下配置:

```

[oslo_messaging_notifications]
driver = messagingv2

```

2) Cinder, 编辑 cinder.conf 中加入以下配置:

```

[oslo_messaging_notifications]
driver = messagingv2

```

3) Heat, 编辑 heat.conf 中加入以下配置:

```

[oslo_messaging_notifications]
driver = messagingv2

```

4) Neutron, 编辑 neutron.conf 中加入以下配置:

```

[oslo_messaging_notifications]
driver = messagingv2

```

5) Sahara, 编辑 sahara.conf 中加入以下配置:

```

[DEFAULT]
enable_notifications=true

```

```
[oslo_messaging_notifications]
driver = messagingv2
```

6) Swift, 首先需要在 Keystone 中建立名为 ResellerAdmin 的角色 (Role), 把这个角色赋予 Ceilometer。此外还需要在 Swift Proxy 配置中加入 Ceilometer 中间件。

```
# 赋予 Ceilometer ResellerAdmin 角色
$ keystone role-create --name=ResellerAdmin
+-----+-----+
| Property | Value |
+-----+-----+
| id       | 462fa46c13fd4798a95a3bfbe27b5e54 |
| name     | ResellerAdmin |
+-----+-----+

$ keystone user-role-add --tenant_id $SERVICE_TENANT \
                        --user_id $CEILOMETER_USER \
                        --role_id 462fa46c13fd4798a95a3bfbe27b5e54

# 编辑 proxy-server.conf 文件, 加入如下内容
[filter:ceilometer]
topic = notifications
driver = messaging
url = rabbit://stackrabbit:openstack1@10.0.2.15:5672/
control_exchange = swift
paste.filter_factory = ceilometermiddleware.swift:filter_factory
set log_level = WARN
# 在 proxy-server 前把 ceilometer 中间件加入
[pipeline:main]
pipeline = catch ..... ceilometer proxy-server
```

7) Nova, 修改 nova.conf 文件, 加入以下配置。

```
[DEFAULT]
instance_usage_audit=True
instance_usage_audit_period=hour
notify_on_state_change=vm_and_task_state

[oslo_messaging_notifications]
driver=messagingv2
```

- 编辑 Ceilometer 的配置文件和 Pipeline 文件。
- 启动 Ceilometer 各项服务。

针对不同场景下的 Ceilometer 应用, 用户可以启动不同的 Ceilometer 服务, 如表 9-5 所示。

表 9-5 Ceilometer 中的服务

目 的	需要启动的服务
处理用户 API 请求	ceilometer-api
获取查询测量值	ceilometer-api ceilometer-polling --polling-namespaces central ceilometer-polling --polling-namespaces compute ceilometer-polling --polling-namespaces ipmi ceilometer-agent-notification ceilometer-collector

用户如果想部署多个 `ceilometer-agent` 需要在配置文件 `ceilometer.conf` 中配置以下选项：

```
[coordination]
# 使用 zookeeper 作为 tooz 的 backend
backend_url=kazoo://<ip of zookeeper>
# 或者使用 memcached 作为 tooz 的 backend
#backend_url=memcached://<ip of memcached server>
```

`ceilometer central polling agent` 可以部署在任意的节点上。

`ceilometer compute polling agent` 需要部署在运行 `nova-compute` 的节点上。

`ceilometer ipmi polling agent` 需要部署在支持 IPMI 的节点上，如果这个节点同时被 `Ironic` 管理，则 `Ironic` 的配置项里必须保证 `conductor.send_sensor_data` 的值为 `false`。否则由于 `Ironic` 会向 `ceilometer` 发送 `notification event`，会出现重复的测量取样。

3. 配置

对于 `Ceilometer` 来说，除了像其他 `OpenStack` 项目一样配置 `ceilometer.conf` 文件之外，还需要配置 `pipeline.yaml` 来定义 `pipeline`。

关于 `ceilometer.conf` 文件的配置项，用户可以在 `Ceilometer` 源代码目录下运行 “`tox -egenconfig`” 命令，之后修改其产生 `etc/ceilometer/ceilometer.conf.sample` 文件，然后复制至系统的 `/etc/ceilometer` 目录。

具体配置项可以参见文档 <http://docs.openstack.org/newton/config-reference/telemetry.html>。

`Pipeline` 的定义，默认保存在文件 `etc/ceilometer/pipeline.yaml` 中，`ceilometer-api` 和各类 `ceilometer-agent` 都要用到。此文件以 `YAML` 格式定义。

下面我们以一个具体的例子解释如何定义 `Pipeline`：

```
---
sources:
  - name: meter_source
    interval: 600
    meters:
      - "!hardware.*"
```



```

    sinks:
      - meter_sink
- name: disk_source
  interval: 600
  meters:
    - "disk.read.bytes"
    - "disk.read.requests"
    - "disk.write.bytes"
    - "disk.write.requests"
  discovery:
    - "local_instances://"
  sinks:
    - disk_sink
- name: hardware_source
  interval: 60
  meters:
    - "hardware.*"
  resources:
    - snmp://192.168.0.11
  sinks:
    - meter_sink
sinks:
- name: meter_sink
  transformers:
  publishers:
    - notifier://?policy=drop&max_queue_length=512
- name: disk_sink
  transformers:
    - name: "rate_of_change"
      parameters:
        source:
          map_from:
            name: "disk\\. (read|write)\\. (bytes|requests)"
            unit: "(B|request)"
          target:
            map_to:
              name: "disk\\.\\1\\.\\2.rate"
              unit: "\\1/s"
            type: "gauge"
        publishers:
          - notifier://
          - udp://10.0.0.2:1234

```

Pipeline 的定义分成两部分：sources（源）数组和 sinks（目标）数组。sources 数组定义

了 Agent 要获取哪些测量值，sinks 数组定义了这些测量值如何经过转化后发送到不同的接收者。

sources 数组中的每一项可以有如表 9-6 所示的字段。

表 9-6 pipeline source 的定义

字段名	说 明
name	字符串类型，当前 source 名称
interval	整数类型，查询时间间隔（只对 polling agent 有效），单位为秒
meters	数组类型，测量值名称，支持通配符 “!” 和 “*”，“*” 表示匹配任意字符，“!” 的意思是取反。对于不同的 agent 来说，其所支持的合法的测量值名称各不相同，具体名称请参见 http://docs.openstack.org/admin-guide/telemetry-measurements.html
discovery	数组类型，可选，discover URL
resources	数组类型，可选，静态资源 URL
sinks	数组类型，表示当前 source 所获取的测量值交由哪些 sink 去处理

discovery 中定义了 discover URL，discover URL 中的 scheme 部分必须是在以 “ceilometer.discover” 为 namespace 的 entry points 中定义的有效值。discovery 的作用是探测发现需要测量的 endpoint，Polling Agent 会对这些 endpoint 进行轮询。

resources 中定义了静态资源 URL。这些静态资源 URL 一般也是用来表示 Polling Agent 会对其进行轮询的 endpoint。可以把 resources 看成是 discover 返回的结果。

如果 source 中没有指定 discovery 或者 resources，Polling Agent 会根据各个 Pollster 的默认 discovery 来获取需要测量的 endpoint。

sinks 数组中的每一项可以有如表 9-7 所示的字段。

表 9-7 pipeline sink 的定义

字段名	说 明
name	字符串类型，当前 sink 名称
transformers	数组类型，定义了需要对进入这个 sink 处理的测量采样值进行怎样的转换
publishers	数组类型，publisher URL，用来发送测量数据采样

Transformers 数组中的每一项需要有一个 name 字段和一个 parameters 字段。name 字段所支持的值必须是在以 “ceilometer.transformer” 为 namespace 的 entry points 中定义的有效值，用来指明选用哪个 transformer 进行处理。parameters 字段包含的是初始化 transformer 的参数，以字典表示。目前支持的 transformer 类型包括以下几种。

- accumulator: 累积保存多个 sample，然后一起给 publisher 发送。
- delta: 计算当前 sample 和前一个相关 sample 的测量值的变化。所谓相关是指前后两个 sample 都属于同一个资源的相同测量值。（有相同的 resource_id 和 name）。
- rate_of_change: 计算当前 sample 和前一个相关 sample 的测量值的变化率。
- unit_conversion: 可以转换当前 sample 中的测量值，主要用于变换测量值的单位。

● arithmetic: 可以对多个或一个 sample 的测量值或者 metadata 进行数学计算。

publishers 数组中的每一项 URL 的 schema 部分必须是在以 “ceilometer.publisher” 为 namespace 的 entry points 中定义的有效值，用来指明经过 Transformer 转换后的测量取样值会被发送给哪些接收者。目前支持的 publisher 请参考 9.2 节。

上面的例子中，定义了如表 9-8 所示的 3 个 Pipeline。

表 9-8 pipeline 定义示例

source 名	轮询间隔	测量值	sink 名	转换	发布
meter_source	600	所有非 “hardware.” 开头的测量值	meter_sink		通过 notifier publisher 发送到 notificaiton bus
transformers	600	在 local_instance 这个 discover 所发现的 endpoint 上获取 4 种与 disk 相关的测量值	disk_sink	rate_of_change	通过 notifier publisher 发送到 notificaiton bus 通过 udp 发送到 10.0.0.2:1234
publishers	60	向主机 192.168.0.11 以 snmp 的方式查询以 “hardware.” 开头的测量值	meter_sink		通过 notifier publisher 发送到 notificaiton bus

4. 使用

用户可以通过 Ceilometer 所提供的 RESTful API 接口来查询 Ceilometer 所保存的测量值，进行警告器 Alarm 的新建、读取、更新和删除操作。具体 API 接口请参见 <http://docs.openstack.org/developer/ceilometer/webapi/v2.html>。

用户也可以通过 python-ceilometerclient 这个 Ceilometer 的 Python 客户端库，以命令行或编程的方式来和 Ceilometer API Server 进行交互。此 Python 库的代码在 <https://github.com/openstack/python-ceilometerclient>。安装了这个库后，用户也可以运行以下命令查看 Ceilometer 命令行程序的帮助。

```
# 安装 python-ceilometerclient 库
$ pip install python-ceilometerclient
# 查看帮助
$ ceilometer --help
```

9.1.9 插件的开发

Ceilometer 在设计之初，就考虑到了可扩展性的问题，所以整体架构的设计允许开发者开发很多不同类型的插件，可以根据自己的需求实现多个层面的功能扩展。

Ceilometer 利用了 stevedore 来实现插件在运行时发现和动态地载入。Ceilometer 中不同类型的插件需要注册在 setup.cfg 文件中 entry_point 段中不同的 namespace 下，如表 9-9 所示。

表 9-9 Ceilometer 中插件的 namespace

namespace 名称	说 明
ceilometer.notification	Notification listener 插件，具体参见 9.1.4 节
ceilometer.discover	Discovery 插件。Discover 的作用是返回需要轮询的资源 endpoint，这些资源 endpoint 会被传递给 pollster 插件使用。具体参见 9.1.3 节
ceilometer.poll.compute	Ceilometer compute agent 所支持的 pollster 插件。具体参见 9.1.3 节
ceilometer.poll.ipmi	Ceilometer ipmi agent 所支持的 pollster 插件。具体参见 9.1.3 节
ceilometer.poll.central	Ceilometer central agent 所支持的 pollster 插件。具体参见 9.1.3 节
ceilometer.metering.storage	Ceilometer 所支持的用以保存测量值相关信息的数据库后台驱动插件。具体参见 9.1.3 节
ceilometer.compute.virt	Ceilometer compute agent 上 pollster 所支持的 hypervisor inspector 插件。具体参见 9.1.4 节
ceilometer.hardware.inspectors	Ceilometer central agent 中 hardware pollster 所支持的 inspector 插件
ceilometer.transformer	Pipeline transformer 插件
ceilometer.publisher	Pipeline publisher 插件，具体参见 9.1.3 节
ceilometer.dispatcher	Ceilometer collector 所支持的 dispatcher 插件
network.statistics.drivers	Ceilometer central agent 中 network statistic pollster 所支持的 driver 插件

1. Pollster

Pollster 作用是实现对某种测量值的轮询获取。Pollster 运行在 Ceilometer Agent 里的某个线程中，周期性地被调用来获取某种测量值的采样值（Sample）。

所有的 Pollster 都必须是 `ceilometer.agent.plugin_base.PollsterBase` 抽象类的子类，需要实现其中接口。PollsterBase 的相关部分定义如下：

```
@six.add_metaclass(abc.ABCMeta)
class PollsterBase(PluginBase):
    def setup_environment(self):
        """用于初始化 pollster"""
        pass
    ...

    @abc.abstractproperty
    def default_discovery(self):
        """返回 Pollster 默认的 discovery"""

    @abc.abstractmethod
```

```
def get_samples(self, manager, cache, resources):
    """返回 Sample 对象列表
    """
```

开发新的 Pollster 插件时，至少需要实现两个接口。其中 `default_discovery` 方法返回一个 URL 字符串来指明这个 Pollster 所需要使用的 discovery 插件。此处 URL 字符串中 `scheme` 部分用来在 `ceilometer.discover` 的 namespace 中找到对应的 discover 插件，URL 字符串中的其余 `netloc` 和 `path` 部分会被作为参数调用这个对应 discover 插件的 `discover` 方法。如果此 Pollster 不需要使用 discovery 插件，可以返回 `None`。

例如，下面的 `default_discovery` 方法表明此 Pollster 需要使用名为 `endpoint` 的 discovery 插件，调用其 `discover` 方法时 `param` 参数的值为 `“compute”`。

```
def default_discovery(self):
    return 'endpoint:compute'
```

Pollster 中主要需要实现的方法是 `get_samples`。该方法被 Ceilometer Agent 周期性地调用。它所接受的输入参数如表 9-10 所示。

表 9-10 `get_samples()` 参数

输入参数名	说 明
<code>manager</code>	指向其运行的 agent service manager 对象的句柄
<code>cache</code>	字典。 <code>pollster</code> 的具体实现可以在这个字典里保存任何信息。注意，这个 <code>cache</code> 是运行在此 agent 上的各个 <code>pollster</code> 共享的。 <code>cache</code> 的作用是帮助某些 <code>pollster</code> 在同一轮询周期里被重复调用的时候，可以利用已有的信息，提高效率。注意，在新一代轮询周期开始时，此 <code>cache</code> 中的信息会被清空
<code>resources</code>	包含 resource endpoint 的列表。这个列表的内容可能是由此 <code>pollster</code> 的 <code>default discovery</code> 插件所获取的 resource 信息，也可能是 pipeline 中 <code>resources</code> 所指定的静态 resource 信息。不同的 <code>pollster</code> 实现可以对这些 <code>resources</code> 值有不同的解释和使用，一般这里的 <code>resources</code> 用来指明这个 <code>pollster</code> 需要哪些 endpoint 上进行获取测量值的操作。

`get_samples` 返回的是一个包含 `ceilometer.sample.Sample` 对象实例的 iterable。这个 iterable 中包含的 `Sample` 对象实例表示的是某个测量值的此次采样值。

Pollster 中的 `setup_environment` 方法可以用来做一些初始化的工作，一般这些工作是用来检查此 `pollster` 是否可以正常工作的。如果发现此 `pollster` 不能正常工作，可以在 `setup_environment` 方法中抛出异常，这样可以此 `pollster` 不被加载进内存，避免无效的 `pollster`。

根据 `stevedore` 的用法，某个插件的实现需要在 `setuptools` 的 entry point 中注册后才能被发现和载入。所以 Pollster 插件可以根据开发者的需要注册在 `ceilometer.poll.compute`、`ceilometer.poll.ipmi` 和 `ceilometer.poll.central` 这 3 个不同名字的空间下，表示其分别运行在 Compute Agent、IPMI Agent 或者 Central Agent。

一般来说，我们建议每一种 Pollster 的实现只返回一种测量值的 `sample` 实例，并且使用

sample.name 的值作为此 Pollster 插件的注册名称，即使用测量值的名称作为 setup.cfg 文件中的注册名称。这样用户在 pipeline meters 中定义的名称，就能和最终测量值名称一致，方便使用。

2. Notification Listener

Ceilometer 的 Notification Listener 的作用是侦听 Notification Bus 上的由其他 OpenStack 服务所发送的通知消息 (notification event)，然后把其所感兴趣的通知消息转成测量取样值 Sample，交给 Pipeline 进一步处理。Notification Listener 插件运行在 Notification Agent 上。

对于大部分情况来说，开发者一般不需要开发自己的 notification listener，而是可以使用通用的 meter notification listener 来将 notification bus 中的通知消息转化成测量取样值 Sample。开发者一般只需要复制源代码中的 ceilometer/ceilometer/meter/data/meters.yaml 文件，添加新的通知消息类型就可以了。此文件支持 JsonPath 格式定义。管理员可以通过定义 meter_definitions_cfg_file 配置选项来载入开发者新开发的定义文件。

所有的 Notification Listener 插件都必须是 ceilometer.agent.plugin_base.NotificationBase 抽象类的子类，需要实现以下的接口：

```
@six.add_metaclass(abc.ABCMeta)
class NotificationBase(PluginBase):
    @abc.abstractproperty
    def event_types(self):
        return

    @abc.abstractmethod
    def get_targets(self, conf):
        return

    @abc.abstractmethod
    def process_notification(self, message):
        return
```

其中，event_types 接口需要返回一个字符串列表，用来表明此 Notification Listener 插件对哪些类型的通知消息感兴趣。

get_targets 方法返回一个包含 oslo.messaging.Target 对象实例的列表，这些 Target 对象实例指明了需要侦听的 Notification Bus 的信息。get_targets 方法的输入参数 conf 指向 oslo.config.CONF 对象，包含了 Ceilometer 的配置信息。下面是一个具体实例：

```
OPTS = [
    cfg.StrOpt('nova_control_exchange',
               default='nova',
               help="Exchange name for Nova notifications."),
]
```



```

cfg.CONF.register_opts(OPTS)

class ComputeNotificationBase(plugin.NotificationBase):
    @staticmethod
    def get_targets(conf):
        return [oslo.messaging.Target(topic=topic,
                                       exchange=conf.nova_control_exchange)
                for topic in conf.notification_topics]

```

`process_notification` 方法用来把消息通知的内容（通过 `message` 输入参数传入）转化为一系列的测量取样值（`ceilometer.sample.Sample` 对象），并返回给 Notification Agent，Notification Agent 会把这些 Sample 交给 Pipeline 处理。

Notification Listener 插件需要被注册在 `ceilometer.notification` 的 namespace 下。一般来说，我们建议每一种 Listener 的实现只返回一种测量值的 Sample 实例，并且使用 `sample.name` 的值作为此 Listener 插件的注册名称，即使用测量值的名称作为 `setup.cfg` 文件中的注册名称。

3. DB Backend Driver

Ceilometer 中测量值（Sample）可以保存在不同类型的后台数据库中，DB Backend Driver 就是为不同类型的后台数据库提供支持。

如果开发者开发支持测量值的后台数据库类型，则需要继承 `ceilometer.storage.base.Connection` 类，要实现的方法如表 9-11 所示。

表 9-11 `ceilometer.storage.base.Connection` 类方法

方法名	输入参数	说 明
<code>upgrade</code>		实现将后台数据库迁移到最新的 schema 版本定义
<code>record_metering_data</code>	<code>data</code> : 字典的列表，每个字典表示一个 Sample 对象，有下述键值： <code>source</code> , <code>counter_name</code> , <code>counter_type</code> , <code>counter_unit</code> , <code>counter_volume</code> , <code>user_id</code> , <code>project_id</code> , <code>resource_id</code> , <code>timestamp</code> , <code>resource_metadata</code> , <code>message_id</code> , <code>message_signature</code>	将 sample 保存在数据库中
<code>get_resources</code>	<code>user</code> – 字符串, user id <code>project</code> – 字符串, tenant id <code>source</code> – 字符串 <code>start_timestamp</code> – 起始时间戳 <code>start_timestamp_op</code> – 字符串, 起始时间戳比较符	根据输入参数查询 resource, 返回包含 <code>ceilometer.storage.models.Resource</code> 对象实例的列表

续表

方法名	输入参数	说 明
	<p>end_timestamp – 结束时间戳</p> <p>end_timestamp_op – 字符串, 结束时间戳比较符</p> <p>metaquery – 字典类型, 用于查询 resource metadata</p> <p>resource – 字符串, resource id</p> <p>pagination – ceilometer.storage.base.Pagination 对象实例句柄</p>	
get_meters	<p>user – 字符串, user id</p> <p>project – 字符串, tenant id</p> <p>resource – 字符串, resource id</p> <p>source – 字符串</p> <p>metaquery – 字典类型, 用于查询 metadata</p> <p>pagination – ceilometer.storage.base.Pagination 对象实例句柄</p>	根据输入参数查询 meter, 返回包含 ceilometer.storage.models.Meter 对象实例的列表
get_samples	<p>sample_filter – ceilometer.storage.SampleFilter 对象实例句柄</p>	根据输入参数查询 sample, 返回包含 ceilometer.storage.models.Sample 对象实例的列表
get_meter_statistics	<p>sample_filter – ceilometer.storage.SampleFilter 对象实例句柄</p> <p>period – 整数, 分段查询中每个时间段的长短, 单位秒</p> <p>groupby – 字符串数组, 根据哪个字段值分组</p> <p>aggregate – 字典数组, 每个字典包含 func 和 param 两个键值, 分别表示 aggregate 的函数名和参数</p>	根据输入参数查询相应的统计数据, 返回包含 ceilometer.storage.models.Statistics 对象实例的列表
query_samples	<p>filtr_expr: 用来查询的 Filter expression</p> <p>order_by: 用来排序的字段名列表</p> <p>limit: 最多返回数据个数</p>	根据输入对 sample 进行复杂查询, 返回包含 ceilometer.storage.models.Sample 对象实例的列表
get_capabilities		<p>返回类似如下的字典, 表示此 driver 所支持的功能</p> <pre>{ 'meters': { 'pagination': False,</pre>

方法名	输入参数	说 明
		<pre> 'query': { 'simple': False, 'metadata': False, 'complex': False} }, 'resources': { 'pagination': False, 'query': { 'simple': False, 'metadata': False, 'complex': False} }, 'samples': { 'pagination': False, 'groupby': False, 'query': { 'simple': False, 'metadata': False, 'complex': False} }, </pre>
get_capabilities		<pre> 'statistics': {00000000000 'pagination': False, 'groupby': False, 'query': { 'simple': False, 'metadata': False, 'complex': False }, 'aggregation': { 'standard': False, 'selectable': { 'max': False, 'min': False, 'sum': False, 'avg': False, 'count': False, 'stddev': False, 'cardinality': False } } } </pre>

续表

方法名	输入参数	说 明
		<pre> } }, 'events': {'query': {'simple': False}}, } </pre>
get_storage_capabilities		返回类似以下的字典，表示此 driver 对于性能的支持： <pre> { 'storage': { 'production_ready': False, } } </pre>

4. Compute Agent Inspector

Ceilometer Compute Agent 运行在和 Nova Compute 服务相同的机器节点上，用来从 Hypervisor 中通过 Compute Pollster 获取相关的测量值。为了对不同的 Hypervisor 进行支持，Ceilometer 抽象出 Compute Agent Inspector 这一层接口，使 Compute Pollster 对不同的下层 Hypervisor 有了统一的调用接口。

Compute Agent Inspector 插件的实现需要继承并实现 `ceilometer.compute.virt.inspector.Inspector` 类，其中所需要实现的接口如表 9-12 所示。

表 9-12 Inspector 类方法

方法名	输入参数	说 明
inspect_instances		查询当前 hypervisor 上运行的 instance，返回 <code>ceilometer.compute.virt.inspector.Instance</code> 对象列表
inspect_cpus	instance_name – VM instance 的名称	查询某个 instance 的 cpu 个数和 cpu 时间，返回 <code>ceilometer.compute.virt.inspector.CPUStats</code> 对象列表
inspect_cpu_util	instance – VM instance 名称 duration – 在最近 n 秒内进行计算	查询某个 instance 最近 duration 秒内的 CPU 利用率，返回 <code>ceilometer.compute.virt.inspector.CPUUtilStats</code> 对象列表
inspect_cpu_l3_cache	instance – VM instance 名称	查询某个 instance 的 cpu level 3 cache 使用量，返回 <code>ceilometer.compute.virt.inspector.CPUL3CacheUsageStats</code> 对象列表
inspect_vnics	instance_name – VM instance 的名称	查询某个 instance 的虚拟网卡相关统计量，返回 (<code>ceilometer.compute.virt.inspector.Interface</code> , <code>ceilometer.compute.virt.inspector.InterfaceStats</code>) 对象列表

续表

方法名	输入参数	说 明
inspect_vnics_rates	instance- VM instance 的名称 duration - 在最近 <i>n</i> 秒内进行计算	查询某个 instance 最近 duration 秒内的网络利用率, 返回 (ceilometer.compute.virt.inspector.Interface, ceilometer.compute.virt.inspector.InterfaceRateStats) 对象列表
inspect_disks	instance_name - VM instance 的名称	查询某个 instance 的磁盘相关统计量, 返回 (ceilometer.compute.virt.inspector.Disk, ceilometer.compute.virt.inspector.DiskStats) 对象列表
inspect_disks_rates	instance- VM instance 的名称 duration - 在最近 <i>n</i> 秒内进行计算	查询某个 instance 最近 duration 秒内的磁盘利用率, 返回 (ceilometer.compute.virt.inspector.Disk, ceilometer.compute.virt.inspector.DiskRateStats) 对象列表
inspect_memory_usage	instance- VM instance 的名称 duration - 在最近 <i>n</i> 秒内进行计算	查询某个 instance 最近 duration 秒内的内存利用率, 返回 ceilometer.compute.virt.inspector. MemoryUsageStats 对象列表
inspect_memory_resident	instance- VM instance 的名称 duration - 在最近 <i>n</i> 秒内进行计算	查询某个 instance 最近 duration 秒内的常驻内存使用情况, 返回 ceilometer.compute.virt.inspector. MemoryResidentStats 对象列表
inspect_memory_bandwidth	instance- VM instance 的名称 duration - 在最近 <i>n</i> 秒内进行计算	查询某个 instance 最近 duration 秒内的内存带宽统计值, 返回 ceilometer.compute.virt.inspector. MemoryBandwidthStats 对象列表
inspect_perf_events	instance- VM instance 的名称 duration - 在最近 <i>n</i> 秒内进行计算	查询某个 instance 最近 duration 秒内的 perf event 统计值, 返回 ceilometer.compute.virt.inspector. PerfEventsStats 对象列表

Compute Agent Inspector 插件需要被注册在 ceilometer.compute.virt 的 namespace 下。用户在配置文件中通过 hypervisor_inspector 配置项指定所需要采用的 Inspector, 这个配置项的合法值是 Agent Inspector 插件的注册名。

5. Publisher

Ceilometer 的 Publisher 作用是将 Pipeline 中的 Sample 发送给特定的接收者。

Publisher 插件需要继承 ceilometer.publisher.PublisherBase 的抽象类:

```
@six.add_metaclass(abc.ABCMeta)
class PublisherBase(object):
    @abc.abstractmethod
    def publish_samples(self, samples):
        return
    @abc.abstractmethod
    def publish_event(self, events):
```



```
return
```

开发者需要实现 `publish_samples` 方法，将 `Sample` 对象发送给特定的接收者。此方法的输入参数 `samples` 是一个包含了 `ceilometer.Sample` 对象的列表。

开发者同时也需要实现 `publish_events` 方法，将 `Event` 对象发送给特定的接收者。输入参数 `events` 是一个包含了 `ceilometer.event.storage.models.Event` 对象的列表。

`Publisher` 插件需要被注册在 `ceilometer.publisher` 的 namespace 下。用户可以在 `Pipeline` 中定义此 `Pipeline` 需要使用哪些 `Publisher`。

6. Discover

`Ceilometer` 的 `Discover` 作用是获取 `Pollster` 所需要轮询的 `endpoint resource` 定义，并以 `resources` 参数传递给 `Pollster` 的 `get_samples` 方法。

所有的 `Discover` 插件都必须是 `ceilometer.agent.plugin_base.DiscoveryBase` 抽象类的子类，需要实现其中的接口。`DiscoveryBase` 的定义如下：

```
@six.add_metaclass(abc.ABCMeta)
class DiscoveryBase(object):
    @abc.abstractmethod
    def discover(self, manager, param=None):
        """Discover resources to monitor.
        """
```

`Discover` 插件需要实现 `discover` 接口，用来返回一个列表。这个列表中的值一般用来表示 `endpoint` 信息，这些 `endpoint` 信息被以 `resources` 参数传递给 `Pollster` 的 `get_samples` 方法。注意，对 `discover` 接口所返回的值的解释由不同的 `Pollster` 实现来掌握，不同的 `Discover` 插件和 `Pollster` 插件可能在这个列表中支持返回不同类型的数据。`Discover` 接口的输入参数如表 9-13 所示。

表 9-13 `Discover()`参数

输入参数名	说 明
<code>manager</code>	指向其运行的 <code>agent service manager</code> 对象的句柄
<code>param</code>	字符串。包含了 <code>discovery URL</code> 中的 <code>netloc</code> 和 <code>path</code> 部分

`Discover` 插件需要注册在 `ceiloemter.discover` 的 namespace 下。

`Discover` 插件和 `Pollster` 插件之间的对应关系可以由以下两种方式指定。注意，第一种指定方法的优先级高于第二种指定方法，即通过第一种方法指定了对应关系后，第二种方法会无效。

- 在 `Pipeline` 配置文件中指定此 `Pipeline` 中 `Pollster` 所需要的 `Discover`。
- 在 `Pollster` 的实现中，`Pollster` 的开发者可以通过 `default_discovery` 方法来指定此 `Pollster` 需要使用的 `Discover` 插件。

不管是哪种方法，都是通过 Discovery URL 字符串的方式来指定 Discover 插件的，一个 Discovery URL 字符串的格式一般为：

```
<scheme>://<netloc>/<path>
```

其中，scheme 部分被用来查找匹配 Discover 插件在 ceilometer.discover 的 namespace 下注册名，netloc 部分和 path 部分被以 param 参数传递给 Discover 插件的 discover 方法。

9.2 Aodh

在 OpenStack Liberty 版本中，为了适应更灵活的部署方案，Telemetry 社区把 Ceilometer 项目中和报警相关的功能剥离出来，成立了一个新的项目 Aodh，主要是提供基于 Ceilometer 所获取的测量值或者 Event 事件进行报警的功能。

9.2.1 体系结构

Aodh 的基本体系结构如图 9-1 所示，主要由以下几种服务构成，每种服务都是可水平扩展（scale out）的。

- API：主要提供面向用户的 RESTful API 接口服务。
- Alarm Evaluator：用来周期性的检查除了 event 类型之外其他警告器（Alarm）相关的告警条件是否满足。
- Alarm Listener：根据消息总线（Notification Bus）上面的 Event 事件消息，来检查相对应的 event 类型的警告器（Alarm）告警条件是否满足。
- Alarm Notifier：当警告器的告警条件满足时，执行用户定义的动作。

目前 Aodh 支持的警告器类型如表 9-14 所示，注意不同类型所对应的测量数据的存储后台不同。

表 9-14 Aodh 告警器类型

类型名	说 明	相应测量值存储后台
threshold	触发条件是某个符合一定条件的测量值或者其统计值的达到某个固定值，例如当属于租户 xyz 的虚拟机 abc 的最近 10min 的平均 cpu 利用率超过 90% 时	Ceilometer 数据库
combination	多个其他警告器触发条件的与操作（and）或者或操作（or）	Ceilometer 或者 Gnocchi
gnocchi_resources_threshold	类似 threshold，但是其测量值存储在 Gnocchi 中	Gnocchi
gnocchi_aggregation_by_metrics_threshold	触发条件基于对多个测量值的统计值再做统计	Gnocchi

续表

类型名	说 明	相应测量值存储后台
gnocchi_aggregation_by_resources_threshold	触发条件基于对符合条件的多个资源（resource）的测量值进行统计	Gnocchi
event	触发条件是符合条件的某个特定 Event 事件到来	Notification Bus
composite	其他 threshold 类型的警告器的与或者或	Ceilometer 或者 Gnocchi

在 Aodh 中所创建的警告器有如表 9-15 所示的 3 种状态。

表 9-15 Ceilometer 警告器状态

状 态	说 明
insufficient data	还没有足够的测量取样值来判断警告器状态
ok	非触发状态，触发条件不满足
alarm	触发状态，触发条件已满足

对于每一种警告器的状态，用户在新建或者修改警告器的时候，都可以为其设置不同的报警动作。当 Alarm Evaluator 周期性检查警告器状态时，或者当 Alarm Listener 接收到相关的 Event 事件并进行检查后，如果发现当前警告器状态有对应的报警动作，那么它会通过 Alarm Notifier 服务来调用相应的报警动作。Aodh 要求用户所设置的报警动作是符合 URL 格式的字符串，Alarm Notifier 服务会根据这个 URL 字符串解析的结果来执行不同的报警动作。目前 Aodh 所支持的报警动作如表 9-16 所示。

表 9-16 Ceilometer 报警动作

URL	说 明
log://	调用 python login.info 记录
http://<host>/<action> https://<host>/<action>	通过对应的 http/https POST 方式调用用户对应的 restful 接口，POST 方法的 body 包含此警告器所被触发的所有信息
trust+http://trust-id@<host>/<action> trust+https://trust-id@<host>/<action>	首先用 trust-id 与 keystone 进行身份认证，将认证获得的 auth_token 作为 http 头 X-Auth-Token 的值，其他和 http/https 报警动作相同
zaqar://	发送给 OpenStack Zaqar 服务

Aodh 的后台数据库采用 SQL 类型的关系型数据库，目前支持 MySQL/PostgreSQL，数据库里主要存放警告器的定义和状态。

Aodh 的 API 服务向用户提供 RESTful API 接口。用户通过 API Server 来对警告器进行 CRUD 操作（新建/读取/更新/删除）。同时，aodh 其他内部服务也会通过 API Server 来获得要检查的警告器列表。Aodh 支持的 API 如表 9-17 所示。

表 9-17 Aodh API

GET /v2/capabilities	查询此 API 版本所支持的功能
GET /v2/alarms	获得符合查询条件的所有警告器
POST /v2/alarms	新建一个警告器
GET /v2/alarms/(alarm_id)	获得某个特定警告器
PUT /v2/alarms/(alarm_id)	修改某个特定警告器
DELETE /v2/alarms/(alarm_id)	删除某个特定警告器
GET /v2/alarms/(alarm_id)/state	获得某个特定警告器的状态
PUT /v2/alarms/(alarm_id)/state	修改某个特定警告器的状态
GET /v2/alarms/(alarm_id)/history	获得符合查询条件的某个特定警告器变动历史
POST /v2/query/alarms	获得符合用户自定义复杂查询的所有警告器
POST /v2/query/alarms/history	获得符合用户自定义复杂查询的所有警告器变动历史

9.2.2 部署与使用

1. 安装与配置

和其他 OpenStack 项目类似，安装 Aodh 有以下几种不同的方式。

(1) 从 Linux 发行版的安装包安装

通过包管理工具安装是最简单的方式。由于不同的 Linux 发行版有着不同的包管理工具，具体请参见 <http://docs.openstack.org/> 网站上的安装指南。

(2) 使用 Devstack 安装

大部分的开发者都会使用 Devstack 来安装配置开发环境。相比与使用安装包，这种方式可以安装最新的 Aodh 代码。用户可以在 Devstack 的 localrc 配置文件中进行如下设置来安装 Aodh（具体可以参见 <http://docs.openstack.org/developer/aodh/install/development.html>）：

```
[[local|localrc]]
enable_plugin aodh https://git.openstack.org/openstack/aodh.git
```

(3) 手动安装

1) 安装后台数据库，具体代码如下：

```
# 下面以 mysql 为例
# 创建数据库，其中 AODH_DBPASS 是用户设定的数据库密码
$ mysql -u root -p
CREATE DATABASE aodh;
GRANT ALL PRIVILEGES ON aodh.* TO 'aodh'@'%' IDENTIFIED BY 'AODH_DBPASS';

# 创建 aodh 的 keystone 用户
$ openstack user create --domain default --password-prompt aodh
User Password:
```

Repeat User Password:

Field	Value
domain_id	e0353a670a9e496da891347c589539e9
enabled	True
id	b7657c9ea07a4556aef5d34cf70713a3
name	aodh

给 aodh 用户赋予 admin 角色

\$ openstack role add --project service --user aodh admin

创建 aodh service endpoint

\$ openstack service create --name aodh \
--description "Telemetry" alarming

Field	Value
description	Telemetry
enabled	True
id	3405453b14da441ebb258edfeba96d83
name	aodh
type	alarming

\$ openstack endpoint create --region RegionOne \
alarming public http://controller:8042

Field	Value
enabled	True
id	340be3625e9b4239a6415d034e98aace
interface	public
region	RegionOne
region_id	RegionOne
service_id	8c2c7f1b9b5049ea9e63757b5533e6d2
service_name	aodh
service_type	alarming
url	http://controller:8042

\$ openstack endpoint create --region RegionOne \
alarming internal http://controller:8042

Field	Value
-------	-------

enabled	True
id	340be3625e9b4239a6415d034e98aace
interface	internal
region	RegionOne
region_id	RegionOne
service_id	8c2c7f1b9b5049ea9e63757b5533e6d2
service_name	aodh
service_type	alarming
url	http://controller:8042

```
$ openstack endpoint create --region RegionOne \
alarming admin http://controller:8042
```

Field	Value
enabled	True
id	340be3625e9b4239a6415d034e98aace
interface	admin
region	RegionOne
region_id	RegionOne
service_id	8c2c7f1b9b5049ea9e63757b5533e6d2
service_name	aodh
service_type	alarming
url	http://controller:8042

2) 安装源代码和服务，具体代码如下：

```
# 获取代码
$ git clone https://git.openstack.org/openstack/aodh.git
# 安装
$ cd aodh
$ sudo python setup.py install
# 产生配置文件
$ tox -egenconfig
# 复制配置文件
$ mkdir -p /etc/aodh
$ cp etc/aodh/aodh.conf.sample /etc/aodh/aodh.conf
$ cp etc/aodh/api_paste.ini /etc/aodh
# 编辑配置文件/etc/aodh/aodh.conf，修改如下相关选项
[database]
...
connection = mysql+pymysql://aodh:AODH_DBPASS@localhost/aodh
```

```

[DEFAULT]
...
rpc_backend = rabbit
auth_strategy = keystone
[oslo_messaging_rabbit]
...
rabbit_host = controller
rabbit_userid = openstack
rabbit_password = RABBIT_PASS
[keystone_authtoken]
...
auth_uri = http://controller:5000
auth_url = http://controller:35357
memcached_servers = controller:11211
auth_type = password
project_domain_name = default
user_domain_name = default
project_name = service
username = aodh
password = AODH_PASS
[service_credentials]
...
auth_type = password
auth_url = http://controller:5000/v3
project_domain_name = default
user_domain_name = default
project_name = service
username = aodh
password = AODH_PASS
interface = internalURL
region_name = RegionOne

# 创建/升级数据库
$ aodh-dbsync

```

3) 如果要使用事件 Event 类型的警告器, 需要配置 Ceilometer event pipeline, 使得相关 Event 事件可以被发送到消息总线的名字是 alarm.all 的 topic 上, 从而被 Aodh 获取。

```

# 在所有运行 ceilometer-agent-notification 的节点上, 编辑
/etc/ceilometer/event_pipeline.yaml 文件, 例如
$ vim /etc/ceilometer/event_pipeline.yaml
---
sources:
  - name: event_source
    events:

```

```

- "*"
sinks:
- event_sink
sinks:
- name: event_sink
  transformers:
  publishers:
    - notifier://
    - notifier://?topic=alarm.all

```

编辑完 event_pipeline.yaml 文件后，需要重启 ceilometer-agent-notification 服务才能使修改生效

4) 启动 aodh 服务，具体代码如下：

```

# 启动 api 服务
$ aodh-api
# 启动 evaluator 服务
$ aodh-evaluator
# 启动 listener 服务
$ aodh-listener
# 启动 notifier 服务
$ aodh-notifier

```

2. 使用

用户可以通过 Aodh 所提供的 RESTful API 接口来进行警告器 Alarm 的新建、读取、更新和删除操作。具体 API 接口请参见 <http://docs.openstack.org/developer/aodh/webapi/v2.html>。

用户也可以通过 python-aodhclient 这个 Aodh 的 Python 客户端库，以命令行或编程的方式来和 Aodh API Server 进行交互。此 Python 库的代码在 <https://github.com/openstack/python-aodhclient>，安装了这个库后，用户也可以运行如下命令查看 Ceilometer 命令行程序的帮助。

```

# 安装 python-aodhclient 库
$ pip install python-aodhclient
# 查看帮助
$ aodh --help

```

9.2.3 插件的开发

与 Ceilometer 类似，Aodh 也是通过 stevedore 来管理它的插件的。Aodh 中不同类型的插件需要注册在 setup.cfg 文件中 entry_point 段中不同的 namespace 下，如表 9-18 所示。

表 9-18 Aodh 中插件的 namespace

namespace 名称	说 明
aodh.storage	Aodh 后台数据库类型驱动
aodh.alarm.rule	Aodh 不同类型的警告器
aodh.evaluator	Aodh 各类型的警告器在 Alarm evaluator 服务中对应相关操作
aodh.notifier	Aodh 不同类型的报警动作

1. 后台数据库

如果开发者开发支持警告器的后台数据库类型,需要继承 `aodh.storage.base.Connection` 类,并实现这个类中的方法。在 `aodh.storage.base.Connection` 类中可以实现的方法如表 9-19 所示。

表 9-19 aodh.storage.base.Connection 类方法

方法名	输入参数	说 明
upgrade		实现将后台数据库迁移到最新的 schema 版本定义
get_alarms	name – 字符串类型, 警告器名称 user – 字符串类型, 警告器的 user id state – 字符串类型, 警告器状态 meter – 字符串类型, 警告器对应的测量值名称 project – 字符串类型, 警告器的 tenant id enabled – 布尔型, 警告器是否使能 alarm_id – 字符串类型, 警告器的 id alarm_type – 字符串类型, 警告器的类型 severity – 字符串类型, 警告器的等级 exclude – 字典类型, 排除含有此字典中字段/值的警告器 pagination – 对结果进行分页控制	查询符合输入参数的警告器的定义并返回
create_alarm	alarm – aodh.storage.models.Alarm 对象实例句柄	新建一个警告器, 返回新建警告器的对象句柄
update_alarm	alarm – aodh.storage.models.Alarm 对象实例句柄	修改警告器, 返回修改后的警告器的对象句柄
delete_alarm	alarm_id – 字符串, 警告器的 id	删除警告器
get_alarm_changes	alarm_id – 字符串类型, 警告器的 id on_behalf_of – 字符串类型, tenant_id, 表示此 tenant 中可以看到 alarm user – 字符串类型, 警告器的 user id project – 字符串类型, 警告器的 tenant id	查询符合输入参数的警告器的变化历史并返回

续表

方法名	输入参数	说 明
	<p>alarm_type - 字符串类型, 警告器的类型</p> <p>severity - 字符串类型, 警告器的等级</p> <p>start_timestamp - 时间戳类型, 历史变化开始时间</p> <p>start_timestamp_op - 字符串, 'gt'表示大于等于开始时间, 其他表示大于开始时间</p> <p>end_timestamp - 时间戳类型, 历史变化结束时间</p> <p>end_timestamp_op - 字符串, 'le'表示小于等于结束时间, 其他表示小于结束时间</p> <p>pagination - 对结果进行分页控制</p>	
record_alarm_change	alarm_change - aodh. storage.models.AlarmChange 对象实例句柄	记录警告器状态或者内容的变化
query_alarms	<p>filter_expr - 由用户 api 中提交的 json 格式的过滤语句所转换而来的 python 对象</p> <p>orderby - (field, direction) 元组列表, 表明按字段的排序顺序</p> <p>limit - 整数, 最多返回多少 alarm 对象</p>	查询符合复杂查询条件的警告器定义并返回
query_alarm_history	<p>filter_expr - 由用户 api 中提交的 json 格式的过滤语句所转换而来的 python 对象</p> <p>orderby - (field, direction) 元组列表, 表明按字段的排序顺序</p> <p>limit - 整数, 最多返回多少 alarm 对象</p>	查询符合复杂查询条件的警告器的变化历史并返回
get_capabilities		<p>返回类似如下的字典, 表示此 driver 所支持的功能</p> <pre>{ 'alarms': {'query': {'simple': False, 'complex': False}, 'history': { 'query': { 'simple': False, 'complex': False} } }, }</pre>

续表

方法名	输入参数	说 明
get_storage_capabilities		返回类似以下的字典，表示此 driver 对于性能的支持： <pre>{ 'storage': { 'production_ready': False}, }</pre>

2. 开发新的告警动作插件

Alarm notifier 的作用是当警告器的状态符合用户定义条件时，由它执行用户定义在警告器中的相应报警动作。当开发者想开发新类型的告警动作插件时，需要继承 aodh.notifier.AlarmNotifier 的抽象类：

```
@six.add_metaclass(abc.ABCMeta)
class AlarmNotifier(object):
    @abc.abstractmethod
    def notify(self, action, alarm_id, alarm_name, severity, previous,
               current, reason, reason_data):
        return
```

开发者需要实现 notify 方法，此方法根据输入参数执行开发者定义动作。它所接受的输入参数如表 9-20 所示。

表 9-20 notify ()参数

输入参数名	说 明
action	被 python urlsplit 函数解析过的 URL
alarm_id	字符串。警告器的 id
alarm_name	字符串类型，警告器的名称
severity	字符串类型，警告器的等级
previous	字符串类型，警告器状态改变前的状态
current	字符串类型，警告器的当前状态
reason	字符串类型，状态改变原因
reason_data	字典类型，状态改变的而外数据

Notifier 插件需要被注册在 aodh.notifier 的 namespace 下。用户可以在创建警告器时通过指定 ok_actions、alarm_actions 和 insufficient_data_actions 来指定当此警告器处于对应状态是应执行的动作。

3. 如何增加一种新的 Alarm 类型

在 Aodh 中，开发者如果要增加一种新的 alarm 警告器类型，一般情况下，至少需要开发

两个插件。一个是 alarm rule 插件，用来定义此警告器的规则，用户根据此规则通过 RESTful API 来进行警告器的 CRUD 操作。另外一个是对应的 alarm evaluator 插件，此插件作用是根
据用户定义的此类型警告器，进行判断，决定警告器的状态等。

开发一个新的 alarm rule 插件，需要继承 aodh.api.v2.base.AlarmRule 类：

```
class Base(wtypes.DynamicBase):

    def as_dict(self, db_model):
        valid_keys = inspect.getargspec(db_model.__init__)[0]
        if 'self' in valid_keys:
            valid_keys.remove('self')
        return self.as_dict_from_keys(valid_keys)

class AlarmRule(Base):
    """Base class Alarm Rule extension and wsme.types."""
    @staticmethod
    def validate_alarm(alarm):
        pass

    @staticmethod
    def create_hook(alarm):
        """ 此钩子函数，在用户新建此类警告器的时候会被调用，传入参数是
        aodh.api.v2.alarms.Alarm 对象的句柄，开发者可以根据具体情况进行后操作。
        """
        pass

    @staticmethod
    def update_hook(alarm):
        """ 此钩子函数，在用户新建此类警告器的时候会被调用，传入参数是
        aodh.api.v2.alarms.Alarm 对象的句柄，开发者可以根据具体情况进行后操作。
        """
        pass
```

开发者一般需要实现两个函数，一个是 AlarmRule 类里面的 validate_alarm 函数，此函数用来验证用户创建或者修改警告器时输入的参数是否合法的。另外一个 AlarmRule 基类 aodh.api.v2.base.Base 类里面的 as_dict 函数，此函数是返回一个字典，里面包括了用户所定义的此类警告器本身的字段，这个字典最后会被序列化后保存在数据库中。

Alarm rule 插件需要被注册在 aodh.alarm.rule 的 namespace 下。

下面我们以一个例子来演示怎样创建一个新的 alarm rule 插件。此例子里，我们创建一个名为 AlarmPredefinedRule 的 alarm rule 插件。用户在创建此插件对应类型的警告器时，可以指定一个字符串，alarm evaluator 检查此类型警告器时，如果此字符串中包含“error”字样，则发出警告。

```

import wsme
from wsme import types as wtypes

from aodh.api.controllers.v2 import base

class AlarmPredefinedStateRule(base.AlarmRule):
    user_string = wsme.wsattr(wtypes.text)

    def __init__(self, user_string=None):
        user_string = user_string or ''
        super(AlarmPredefinedStateRule, self).__init__(user_string=
                                                    user_string)

    @classmethod
    def validate_alarm(cls, alarm):
        super(AlarmPredefinedStateRule, cls).validate_alarm(alarm)
        if alarm.rule.user_string == '':
            raise base.ClientSideError("user_string can not be empty")

    def as_dict(self):
        return self.as_dict_from_keys(['user_string'])

```

然后编辑 `setup.cfg`, 注册在 `aodh.alarm.rule` 的 namespace 下。

```

# setup.cfg
...

[entry_point]
...
aodh.alarm.rule =
...
    predefined =
aodh.api.controllers.v2.alarm_rules.predefined:AlarmPredefinedStateRule

```

开发一个新的 alarm evaluator 插件, 需要继承 `aodh.evaluator.Evaluator` 类:

```

@six.add_metaclass(abc.ABCMeta)
class Evaluator(object):
    ...
    @abc.abstractmethod
    def evaluate(self, alarm):
        """Interface definition.
        evaluate an alarm
        alarm Alarm: an instance of the Alarm
        """

```

evaluator 插件需要实现 `evaluate` 这个抽象方法。在这个方法中, 开发者需要检查由参数

alarm 传入的警告器报警条件是否满足，如果满足的话需要修改状态，发出警告。

我们使用上面同样的例子来开发对应的 alarm evaluator 插件。

```
from aodh import evaluator

class PredefinedEvaluator(evaluator.Evaluator):
    def evaluate(self, alarm):
        if 'error' in alarm.rule['user_string']:
            state = evaluator.ALARM
        else:
            state = evaluator.OK
            # 保存警告器新的状态，并根据情况调用报警动作
            self._refresh(alarm=alarm,
                          state=state,
                          reason='foo',
                          reason_data={})
```

然后编辑 setup.cfg，注册在 aodh.evaluator 的 namespace 下。

```
# setup.cfg
...

[entry_point]
...
aodh.evaluator =
...
predefined = aodh.evaluator.predefined:PredefinedEvaluator
```

9.3 Gnocchi

Ceilometer 中，每个测量值的采样中都包括了资源 (resource) 的 metadata，这些 metadata 大多数时候都是不改变的。而由于计费审计的需要，Ceilometer 会把所有的采样值都保存在数据库中，这样就有大量重复的 resource metadata 数据也一并保存在数据库中，从而导致了在大数据量的情况下，后端数据库大小增长很快。另外，由于 Ceilometer 在计算统计信息时，是根据保存的所有原始测量采样值计算的，这样导致如果统计中有非常多的采样点的话，计算统计信息的性能不够好。

Gnocchi 项目的提出主要是为了解决 Ceilometer 上述的后端存储性能问题的。它把测量值和资源 resource 分开存储在不同的后端，对重复的 resource_metadata 只保存一份。另外，Gnocchi 并不会把原始的测量值一直保存，而是根据用户定义的每个测量值的 archive_policy，来保存最近的测量值。并且当收到用户发送来的测量值采样时，根据 archive_policy，动态地计算出统计信息。这样，Gnocchi 从这两个方面解决了 Ceilometer 后台数据保存量太大和获取统计值较慢的问题。

Gnocchi 项目的目标是提供一个支持多租户的数据存储,用来保存基于时间序列测量值和资源 resource 信息。这个项目也是 OpenStack Big Tent 下面的项目,但是它的版本发布不同于 OpenStack,有着自己的版本发布节奏。

9.3.1 体系结构

Gnocchi 的基本体系结构如图 9-6 所示。

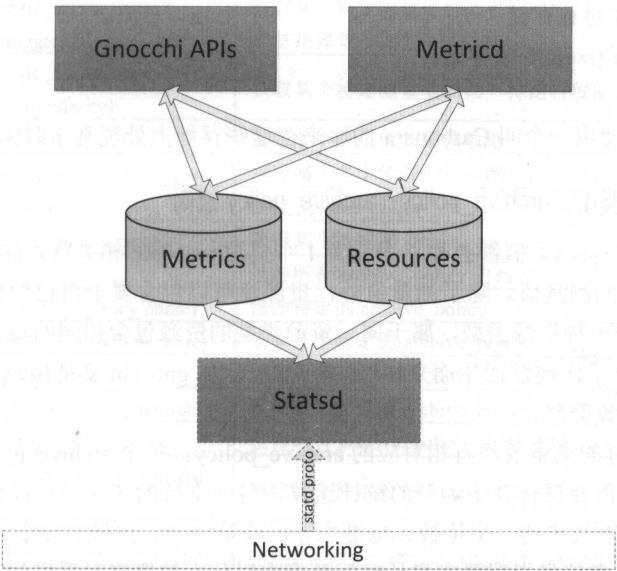


图 9-6 Gnocchi 体系结构

Gnocchi 主要由以下几种服务构成,每种服务都是可水平扩展 (scale out) 的。

- API: 主要提供面向用户的 RESTful API 接口服务。
- metricd: 异步处理 API 服务或者 statd 服务接收到的数据,比如计算统计值、清理过期的测量值等。
- statd: 支持 statd 协议,可以从网络上接受其他的服务通过 statd 协议发送过来的测量值。

1. 数据后端

Gnocchi 使用两种不同的数据后端来保存不同类型的数据:对于基于时间序列的测量值采样数据,gnocchi 内部使用 storage driver 来保存;对于资源 resource 以及测量值的索引操作,使用 index driver 来保存。

storage driver 后端保存的是测量值采样,每个测量值的采样数据只包含两项信息,一个时

间戳，一个是采样值。storage driver 获得采样值后，动态地就会根据此测量值的 archive_policy 把需要的统计信息计算出来。

index driver 后端用来保存所有的资源，包括它们的类型和属性。同时，它也负责建立资源和测量值之间的关系。

目前 index driver 后端只支持 SQL 类型的关系数据库，即 MySQL 或者 PostgreSQL。

storage driver 后端支持多种不同类型的存储，其中包括：

- 文件系统存储。
- 亚马逊 S3 对象存储。
- OpenStack Swift 对象存储。
- Ceph。

storage driver 使用一个叫 Carbonara 的库来在这些存储上处理基于时间序列的测量数据。

2. 资源，资源类型，archive_policy，archive_policy 规则

Gnocchi 中每一种测量值都会和 0 个或者 1 个资源 resource 相关联，同一个资源 resource 可以包括多个不同的测量值。除了测量值外，资源还可以包含属于自己的特有属性。每种资源 resource 都属于一种资源类型，属于同一资源类型的资源包含相同的特有属性。所有资源类型都是从 generic 资源属性派生而来。从版本 3.0.0 开始，gnocchi 支持用户通过 RESTful API 接口创建自己的资源类型。

Gnocchi 中所有的测量值都有相对应的 archive_policy。每个 archive_policy 中包含了两类信息，一类信息是需要进行哪些类型的统计运算，目前支持的统计运算操作有最大值、最小值、求和、计数、算数平均、中位数、标准方差、找第一个、找最后一个和 Npct (N 为 1~100 的整数)。另一类信息指明要在多长时间跨度内以什么粒度来对采样信息计算统计值，用户可以选取三元组 (points、granularity 和 timespan) 中的任意两个数据来指定时间跨度。三元组内元素的含义如表 9-21 所示。

表 9-21 archive_policy 时间序列跨度含义

名 称	说 明
points	总计多少个采样点参与运算
granularity	最小采样时间粒度
timespan	时间跨度

Archive_policy 规则定义了 archive_policy 和测量值之间的对应关系。比如用户可以定义所有名字以 disk.io.开头的测量值都对应使用名字为 low 的 archive_policy。

3. Gnocchi API server

Gnocchi 的 API 服务向用户提供 RESTful API 接口。用户通过 API Server 来对资源 resource、资源类型、测量值、archive_policy、archive_policy 规则进行 CRUD 操作（新建/读取/更新/删

除)。同时还能通过 API sever 来递交测量值采样, 和获取测量值的统计信息。具体 API 如表 9-22 所示。

表 9-22 Gnocchi API

POST /v1/metric	新建测量值
GET /v1/metric/(metric_id)	获得一个特定的测量值
GET /v1/metric	获取所有测量值的列表
POST /v1/metric/(metric_id)/measures	发送某个特定测量值的采样数据
GET /v1/metric/(metric_id)/measures? aggregation=(aggregation_method)	获取某个特定测量值的所有粒度的某一类统计信息（默认统计信息是算数平均）
GET /v1/metric/(metric_id)/measures? ?granularity=(granularity)	获取某个特定测量值的某一特定粒度的算数平均数统计信息
POST /v1/batch/metrics/measures	发送多个测量值的采样数据
POST /v1/batch/resources/metrics/measures	发送属于多个资源的不同测量值的采样数据
POST /v1/archive_policy	新建 archive_policy
GET /v1/archive_policy/(policy name)	获得某个特定的 archive_policy
GET /v1/archive_policy	获得所有 archive_policy 的列表
PATCH /v1/archive_policy/(policy name)	修改特定的 archive_policy
DELETE /v1/archive_policy/(policy name)	删除某个特定的 archive_policy
POST /v1/archive_policy_rule	新建 archive_policy 规则
GET /v1/archive_policy_rule/(rule name)	获得某个特定的 archive_policy 规则
GET /v1/archive_policy_rule	获得所有 archive_policy 规则的列表
DELETE /v1/archive_policy_rule/(rule name)	删除某个特定的 archive_policy 规则
POST /v1/resource/generic	新建 generic 类型的资源
POST /v1/resource/(resource type)	新建 resource_type 类型的资源
GET /v1/resource/generic/(resource_id)	获得属于 generic 类型的某个特定资源
PATCH /v1/resource/(resource type) /(resource_id)	修改属于 resource_type 类型的某个特定资源
GET /v1/resource/(resource type) /(resource_id)/history	获得属于 resource_type 类型的某个特定资源的修改历史
DELETE /v1/resource/generic/(resource_id)	删除特定资源
DELETE /v1/resource/generic	批量删除资源
GET /v1/resource/generic/(resource_id)/metric/(metric name)/measures	获取和某个特定资源相关联的特定测量值的所有统计值
POST /v1/resource/generic/(resource_id)/metric	新增一个和某特定资源相关联的测量值
POST /v1/resource_type	新建资源类型
GET /v1/resource_type/(resource type name)	获取特定的资源类型定义
GET /v1/resource_type	获取所有资源类型列表

DELETE /v1/resource_type/(resource type name)	删除特定的资源类型
PATCH /v1/resource_type/(resource type name)	修改特定的资源类型
POST /v1/search/resource/(resource type)	搜索符合条件的属于特定类型的资源
POST /v1/search/resource/(resource type)? history=true	搜索符合条件的属于特定类型的资源修改历史记录
POST /v1/search/metric?metric_id=(metric_id)	搜索符合条件的特定测量值的统计值
GET /v1/aggregation/metric?metric=(metric_id1) &metric=(metric_id2)&aggregation=(aggregation_method)	对多个测量值的统计信息再次计算统计值
GET /v1/capabilities	获取 gnocchi 当前支持统计运算操作
GET /v1/status	获取 gnocchi 当前的运行状态

9.3.2 部署与使用

1. 安装与配置

由于 Gnocchi 和 OpenStack 有着不同的发布周期，一般采用以下几种方法安装 Gnocchi。

(1) 使用 Devstack 安装

大部分的开发者都会使用 Devstack 来安装配置开发环境。相比与使用安装包，这种方式可以安装最新的 Gnocchi 代码。用户可以在 Devstack 的 localrc 配置文件中设置如下项来安装 Gnocchi（具体可以参见 <http://docs.openstack.org/developer/aodh/install/development.html>）：

```
[[local|localrc]]
enable_plugin gnocchi https://git.openstack.org/openstack/gnocchi.git master
```

(2) 手动安装

安装前，用户需要考虑选取 storage driver 后台存储大小。按最坏情况考虑，对于每一个参与运算的采样点，每种统计运算操作会占用 8 个字节。用户需要根据实际情况估计后台存储的大小要求。

```
# 获取代码
$ git clone https://git.openstack.org/openstack/gnocchi.git
# 安装 gnocchi 自己的代码
$ cd gnocchi
$ pip install -e .
# 根据数据后端类型的选取和不同的功能，安装对应的不同依赖包，例如
$ pip install -e .[postgresql,ceph,ceph_recommended_lib]
# 产生配置文件
$ oslo-config-generator
```

```
--config-file=/etc/gnocchi/gnocchi-config-generator.conf
--output-file=/etc/gnocchi/gnocchi.conf
# 编辑配置文件, 部分配置选项参见表 9-23
$ vim /etc/gnocchi/gnocchi.conf
# 初始化数据后端
$ gnocchi-upgrade
# 运行 gnocchi 服务
$ gnocchi-api
$ gnocchi-metricd
```

表 9-23 Gnocchi 部分配置参数

配置参数名称	说 明
storage.driver	storage driver 后端
indexer.url	指定 index driver 的 URL
storage.coordination_url	Openstack tooz 库的后台, 用于多个 gnocchi api 和 gnocchi metricd 的协同工作
storage.file_*	storage driver 使用文件系统作为后端时的相关配置
storage.s3_*	storage driver 使用亚马逊 S3 作为后端时的相关配置
storage.swift_*	storage driver 使用 Swift 作为后端时的相关配置
storage.ceph_*	storage driver 使用 ceph 作为后端时的相关配置

Gnocchi 目前不同功能的安装依赖包如表 9-24 所示。

表 9-24 Gnocchi 可选功能安装依赖包

功能名称	说 明
keystone	提供 keystone 身份验证功能
mysql	index driver 后台使用 mysql 作为数据库
postgresql	index driver 后台使用 postgresql 作为数据库
file	storage driver 使用文件系统作为后端
s3	storage driver 使用亚马逊 S3
swift	storage driver 使用 Swift 作为后端
ceph	storage driver 使用 ceph 作为后端的通用依赖库
ceph_recommended_lib	Ceph 版本≥0.80 的支持
doc	产生文档需要的依赖库
test	测试运行依赖库

2. 使用

用户可以通过 Gnocchi 所提供的 RESTful API 接口来使用 Gnocchi。具体 API 接口请参见 <http://docs.openstack.org/developer/gnocchi/rest.html>。

用户也可以通过 python-gnocchiclient 这个 Gnocchi 的 Python 客户端库, 以命令行或编程的方式来和 Gnocchi API Server 进行交互。此 Python 库的代码在 <https://github.com>。

com/openstack/python-gnocchiclient。安装了这个库后，用户也可以运行如下命令查看 `gnocchi` 命令程序的帮助。

```
# 安装 python-gnocchiclient 库/  
$ pip install python-gnocchiclient  
# 查看帮助  
$ gnocchi --help
```

9.4 Panko

在最新的 OpenStack Newton 版本中，Telemetry 社区把 Ceilometer 关于 Event 事件部分的 API 移到了 Panko 项目中。目前，Event 事件的产生和保存仍旧是由 Ceilometer 负责，但是对于从后台数据库中读取 Event 事件的 API service 被移到了 Panko 项目中（Ceilometer API 中与 Event 相关的 API 目前还保留着）。目前 Panko 项目中只有 API server 功能，此项目还处于初步发展演进阶段。API 具体参见 <http://docs.openstack.org/developer/panko/webapi/v2.html>。

物理机管理

在云计算蓬勃发展的今天，虚拟化技术作为云计算的基石得到了广泛的应用，尤其是虚拟机的特性很好地切合了云计算的需求，在裸机上部署和运行工作负载的方式似乎渐渐被大家遗忘，但在把工作负载往云上部署和迁移的过程中，大家也意识到在裸机上运行负载的方式同样必不可少，具体到以下几个方面的应用：

- 高性能计算，采用物理机直接运行负载可以减少虚拟化层的消耗，提高 I/O 的负载。
- 需要用到物理设备资源的计算任务，而这些物理资源不能被虚拟化出来。
- 数据库应用，有些数据库应用对内存及 I/O 的要求高，在虚拟化上运行性能很差，不能满足应用要求。
- 裸机托管，客户把自己的物理机器放到云服务商的云环境中，这就需要该云环境具备裸机云的管理能力。
- 云环境的部署，比如 OpenStack TripleO 项目利用 OpenStack 来部署 OpenStack，需要利用已有 OpenStack 环境的裸机服务（Bare Metal，也就是 Ironi 项目）去部署裸机。

另一方面，通过自动化工具来完成系统部署、系统参数配置、软件安装等一系列工作，已经是系统管理员的必备技能。随着云计算互联网时代的到来，其所拥有的快速迭代与高可扩展性（Scalability）又对系统管理员提出了更高的要求。比如部署周期，在一些极端案例中，比如 Flickr 的业务部门，每天需要部署多达 10 次以上，仅仅是之前的自动化工具就已经无法完成任务。然后是部署单位，需要摆脱以往以单节点为部署单元的方式，进而转向以云为部署单元，即一次性部署多个不同的节点，从而搭建一个可立刻上线的云。

最后，在混合云大行其道的今天，云环境中对虚拟机和物理机的部署和管理需要统一的接口，SoftLayer、Rackspace、Oracle 以及 Internap 等云环境也都已经实现了 BareMetal 的云管理。物理机和虚拟机的管理有很多非常相似的地方，比如它们都需要开机/关机、部署/安装、添加/删除等，因此早期对物理机的管理是通过 Nova 中的 BareMetal 驱动来完成的，这样避免了很多重复的实现。

OpenStack Ironi 项目的出现正是为了解决以上问题，应用于 OpenStack 中的裸机管理和部署。

10.1 Ironi 体系结构

Ironi 生态由一系列项目组成，包括 Ironi 本身以及 ironi-python-agent、ironi-inspector

等。

Ironic 项目本身的体系结构如图 10-1 所示, 主要由 ironic-api 和 ironic-conductor 两个服务组成, ironic-api 提供了 RESTful API 服务, 是访问并使用 Ironic 所提供服务的唯一途径, ironic-api 对数据库的访问主要在创建资源的时候; ironic-conductor 负责与数据库进行交互, 完成实际的工作。

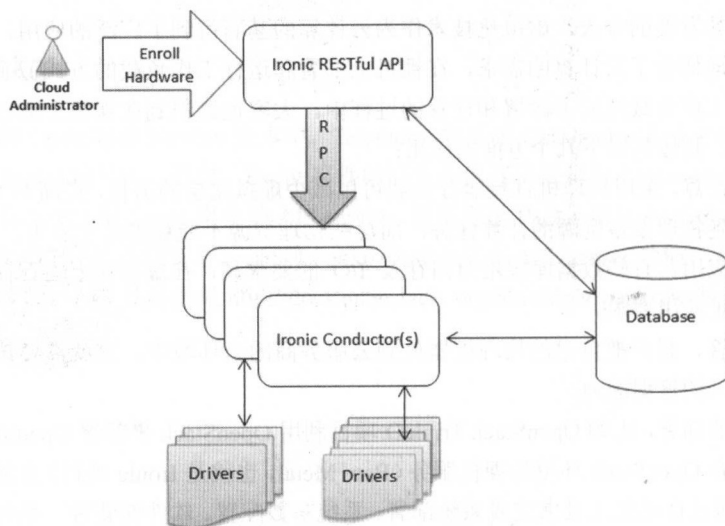


图 10-1 Ironic 体系结构

Ironic 每个交互的类型都是实际硬件的操作, 比如电源、启动、部署等, 根据物理硬件的不同, Ironic 提供了一个驱动的框架来进行支持。目前已经实现的驱动有 PXE-IPMITool、PXE-IPMINative、PXE-SSH 等。

ironic-api 与 ironic-conductor 之间通过 RPC 进行通信。Ironic API 的执行过程与使用方式与之前介绍的其他项目类似, Conductor 的理念也类似于 Nova 的 Conductor 服务, 用于提高数据库的安全性和并发性。ironic-api 接收到用户的 RESTful 请求后, 最终会通过 RPC 调用 Conductor 的方法完成实际的工作, 而 Conductor 又会根据配置使用具体的驱动, 比如使用 PXE 去完成真正的部署。

Ironic 服务有以下两种使用模式:

- 和 Nova 服务一起, 作为 Nova 的一个 virtDriver 来使用, Nova 把裸机当作虚拟机来对待, 能够利用 Nova 服务中的多租户、调度、配额管理等功能, 构成为真正的裸机云。
- Standalone 单独使用模式, 没有多租户, 没有调度, 没有配额管理等功能, 更多的是一种裸机的云部署工具。

图 10-2 所示为与 Nova 一起工作时 Ironic 服务与 OpenStack 其他组件的交互关系。

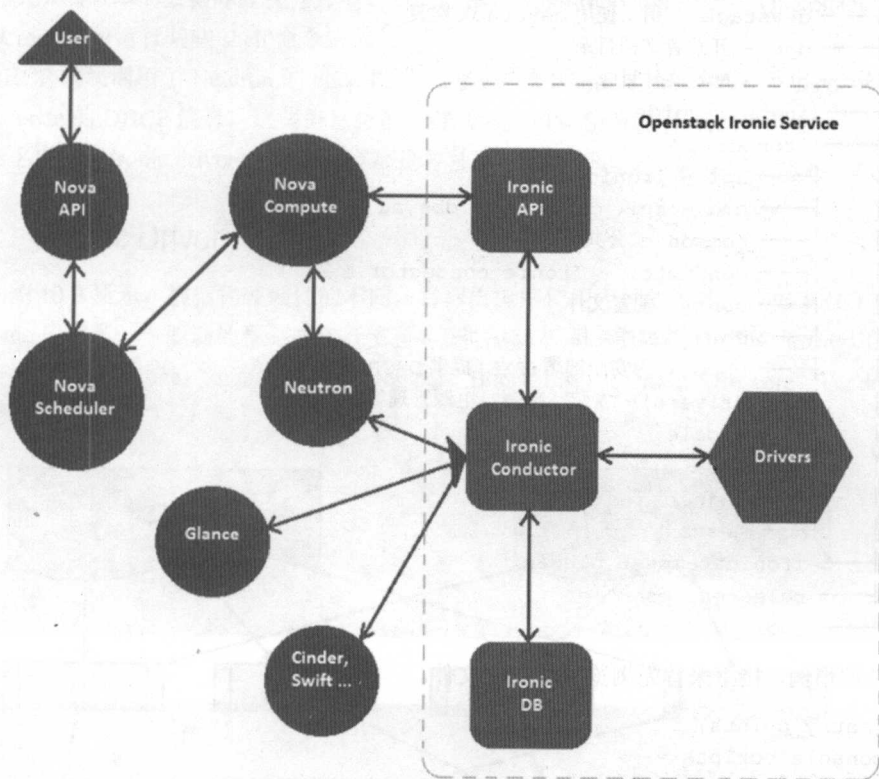


图 10-2 Ironic 与其他服务的关系

- 用户通过 Nova API 发出 boot instance 请求，Nova Scheduler 通过消息队列拿到这个请求。
- Nova Scheduler 根据 Flavor 中的 extra_specs 信息，比如 “cpu_arch ” “baremetal:deploy_kernel_id” “baremetal:deploy_ramdisk_id” 来选择合适的裸机节点。
- 之后，根据 Scheduler 选出来的节点，Nova Compute 服务通过 Ironic virtDriver 启动一个任务，通过 Ironic API 查找该节点的信息，并预留该节点。
- 把请求部署的镜像从 Glance 下载下来并保存到 Ironic Conductor 的本地磁盘。
- 调用 Neutron API 来做网卡绑定并设置 Neutron 中的 DHCP 端口来支持 PXE/TFTP 服务。
- Ironic virtDriver 通过 Ironic API 发出部署请求给 Ironic Conductor。
- Ironic Conductor 调用对应的 Ironic 驱动来完成部署。

Ironic 源码结构如下：

- |— api-ref
- |— devstack - 用于使用 devstack 安装
- |— doc - 开发者文档目录
- |— etc - 配置文件目录
- |— install-guide
- |— ironic
 - |— api - IroniC API 实现
 - |— cmd - api、conductor、dbsync 命令
 - |— common - 公共代码
 - |— conductor - IroniC conductor 服务
 - |— conf - 配置文件
 - |— db - 数据库操作
 - |— dhcp - 为安装部署 (PXE) 提供 DHCP 服务
 - |— drivers - 安装、部署、电源管理等驱动
 - |— locale
 - |— nova
 - |— objects
 - |— tests
- |— ironic_tempest_plugin
- |— releasenotes
- |— tools

依照惯例，接下来首先浏览 `setup.cfg` 文件。

```
[entry_points]
console_scripts =
    ironic-api = ironic.cmd.api:main
    ironic-dbsync = ironic.cmd.dbsync:main
    ironic-conductor = ironic.cmd.conductor:main
    ironic-rootwrap = oslo_rootwrap.cmd:main

ironic.dhcp =
    neutron = ironic.dhcp.neutron:NeutronDHCPApi
    none = ironic.dhcp.none:NoneDHCPApi

ironic.drivers =
    agent_amt = ironic.drivers.agent:AgentAndAMTDriver
    agent_iboot = ironic.drivers.agent:AgentAndIBootDriver
    agent_ilo = ironic.drivers.ilo:IloVirtualMediaAgentDriver
    agent_ipmitool = ironic.drivers.agent:AgentAndIPMIToolDriver
.....
```

命名空间“`console_scripts`”中的每一项都表示一个可执行的脚本，这些脚本在部署时会被安装。对于 IroniC 来说，我们可以看到，除了两个主要的服务 API 与 Conductor，还有两个

辅助的工具 `ironic-dbsync` 与 `ironic-rootwrap`，其中，`ironic-dbsync` 用于创建数据库的 `schema` 以及负责从旧版本的数据库迁移与升级到新的版本，`ironic-rootwrap` 用于在 OpenStack 运行过程中以 `root` 身份运行某些 `shell` 命令。

DHCP 模块调用了 Neutron 的 `dhcp` 接口，通过修改 Neutron `port` 中 `extra_dhcp_opts` 参数来设置 `node` 的 DHCP 信息，这些信息会被 PXE 启动和 iSCSI 部署用到。

命名空间 “`ironic.drivers`” 中的内容都是各种驱动的入口。

10.1.1 Ironic Driver

如图 10-3 所示，根据物理硬件的不同，目标机能够使用不同的方式（比如 PXE）进行部署，Ironic 提供了一个驱动的框架来进行支持，每种驱动基本需要实现 `Deploy`、`Power`、`Console` 等几类核心功能。此外，还有其他 `Management`、`Boot`、`Inspect`、`Raid`、`Rescure`、`Vendor` 等几种标准功能。

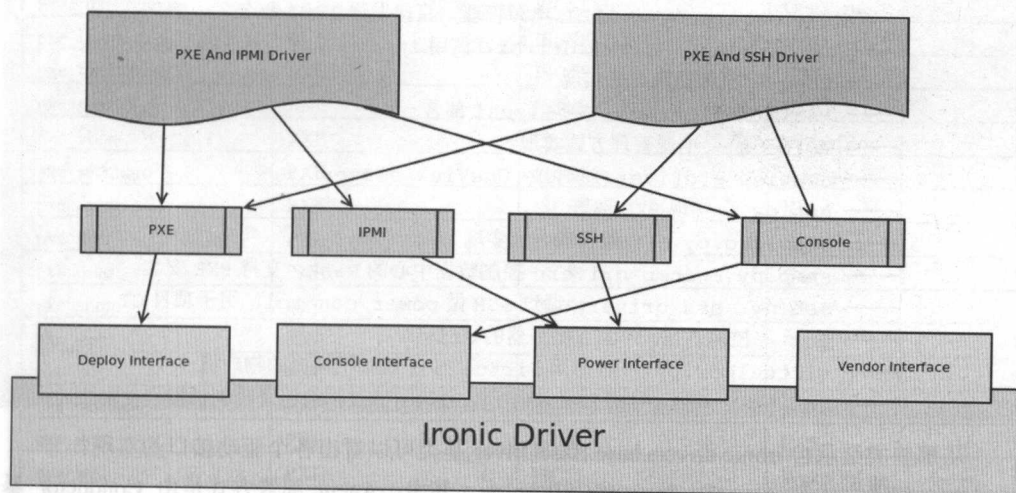


图 10-3 Ironic 驱动框架

其中，`Deploy` 接口实现机器镜像部署，`Power` 接口实现机器的开机、关机、重启等操作，`Console` 用来获取/操作机器的 `console` 接口，`Management` 接口实现机器启动设备的管理，`Inspect` 实现机器 `Capacity` 的探测，`Raid` 实现机器存储 `Raid` 功能的设置，`Rescure` 可以让机器进入安全模式启动，`Vendor` 把各个 `vendor` 特定的功能透传给驱动。

Ironic 驱动的源码位于 `ironic/drivers` 目录：

- `agent.py` - `agent` 部署方式+各种机器控制组成的驱动
- `base.py` - 驱动接口的基类
- `drac.py` - `PXE``Drac` 驱动，使用 `PXE` 部署方式


```

├── fake.py - 测试用驱动
├── ilo.py - HPE ProLiant servers 控制接口, 支持 PXE/iSCSI/Agent
├── irmc.py - FUJITSU PRIMERGY server 控制接口, 支持 PXE/iSCSI/Agent
├── oneview.py - oneview 电源管理相关驱动
├── pxe.py - 以 PXE 为启动方式的驱动
├── modules - 各种核心和标准驱动模块的实现目录
│   ├── agent_base_vendor.py
│   ├── agent_client.py
│   ├── agent.py - Agent 部署方式实现
│   ├── amt - Active Management Technology driver, 用于 desktop
│   ├── cimc - 控制 Cisco UCS C series servers 的 Driver
│   ├── drac - 控制 FUJITSU PRIMERGY server, 支持 PXE/iSCSI/Agent
│   ├── iboot.py - dataprobe iboot driver, 支持 PXE 和 Agent 部署
│   ├── ilo - ilo 电源管理实现
│   ├── image_cache.py - 从 glance 获取启动镜像并缓存到这里
│   ├── inspector.py - capabilities 探测
│   ├── ipminative.py - IPMI 电源管理, 直接发送 IPMI 命令
│   ├── ipmitool.py - IPMI driver, 调用 ipmitool 库发送 IPMI 命令
│   ├── irmc - 电源管理方式实现
│   ├── iscsi_deploy.py - 实现 iscsi 部署
│   ├── msftocs - 电源管理方式实现
│   ├── oneview - driver, 控制 HP OneView server
│   ├── pxe.py - 实现 PXE 部署
│   ├── seamicro.py - 电源管理方式实现
│   ├── snmp.py - snmp driver, 控制数据中心的 Rack, 支持 PXE 安装
│   ├── ssh.py - ssh driver, 通过 SSH 做 power control, 用于项目 CI
│   ├── ucs - 控制 Cisco UCS 服务器的 driver
│   ├── virtualbox.py - 控制用 virtualbox 虚拟机模拟的物理机
│   └── wol.py - Wake-on-LAN (WoL) driver 通过网络消息控制机器

```

从那些类继承自 `ironic.drivers.base.BaseDriver` 基类可以看出各个驱动接口的实现。

- 部署方式有 `AgentDeploy`、`ISCSIDeploy`。其中, `Agent` 部署方式是由 `Conductor` 暴露出一个 `image` 的临时 `Swift URL`, 之后 `IPA (Ironic-Python-Agent)` 会处理所有余下的 `deploy` 工作, 即从 `Swift` 上下载该 `image` 并放到机器上安装部署。`ISCSI` 部署通过 `iSCSI` 挂载物理节点上的硬盘到控制节点来完成镜像复制。
- 启动方式有 `IloVirtualMediaBoot`、`IRMCVirtualMediaBoot`、`PXEBoot`。启动方式基本都是 `PXEBoot`, 其他两种启动方式是在某特定服务器支持的特殊的启动方式。
- 电源管理方式有 `AMTPower`、`DracPower`、`IBootPower`、`IloPower`、`NativeIPMIPower`、`IPMIPower`、`IRMCPower`、`MSFTOCSPower`、`OneViewPower`、`SNMPPower`、`SSHPower`、`VirtualBoxPower`、`WakeOnLanPower`。电源管理方式的多样性也是由于不同主要服务器厂商的各自有电源管理方式。比较通用的是 `IPMI`。

各个驱动启动、部署和电源管理方式如表 10-1 所示。

表 10-1 驱动的启动、部署和电源管理方式

驱 动	启 动	部 署	电源管理
agent_amt	PXE	Agent	AMT
agent_iboot	PXE	Agent	IBoot
agent_ilo	IloVirtualMedia	Agent	Ilo
agent_ipmitool	PXE	Agent	IPMI
agent_irmc	IRMCVirtualMedia	Agent	IRMC
agent_pxe_oneview	PXE	Agent	OneView
agent_pyghmi	PXE	Agent	IPMI
agent_ssh	PXE	Agent	SSH
agent_vbox	PXE	Agent	VirtualBox
agent_ucs	PXE	Agent	UCS
agent_wol	PXE	Agent	WakeOnLan
iscsi_ilo	IloVirtualMedia	ISCSI	Ilo
iscsi_irmc	IRMCVirtualMedia	ISCSI	IRMC
iscsi_pxe_oneview	PXE	ISCSI	OneView
pxe_ipmitool	PXE	ISCSI	IPMI
pxe_ipminative	PXE	ISCSI	IPMI
pxe_ssh	PXE	ISCSI	SSH
pxe_vbox	PXE	ISCSI	VirtualBox
pxe_seamicro	PXE	ISCSI	seamicro
pxe_iboot	PXE	ISCSI	IBoot
pxe_ilo	PXE	ISCSI	Ilo
pxe_drac	PXE	ISCSI	Drac
pxe_snmp	PXE	ISCSI	SNMP
pxe_irmc	PXE	ISCSI	IRMC
pxe_amt	PXE	ISCSI	AMT
pxe_msftocs	PXE	ISCSI	MSFTOCS
pxe_ucs	PXE	ISCSI	UCS
pxe_wol	PXE	ISCSI	WakeOnLan
pxe_iscsi_cimc	PXE	ISCSI	Cim_power
pxe_agent_cimc	PXE	Agent	Cim_power

这里以 PXE-IPMI 驱动为例来进行分析。

IPMI 是一项应用于服务器管理系统设计的标准，由 Intel、惠普、Dell 与 NEC 于 1998 年共同提出，用户可以使用 IPMI 监控服务器的温度、电压、电源状态等，此接口标准有助于在不同类服务器系统硬件上实施系统管理。

PXE (Pre-boot ExecutionEnvironment) 是由 Intel 设计的协议，它可以使计算机通过网络而不是从本地硬盘、光驱等设备启动。现代的网卡，一般都内嵌支持 PXE 的 ROM 芯片。PXE 的工作原理就是管理员通过 IPMI 远程重启目标机，目标机会将网卡里的 PXE Client 载入内存，使得目标机在没有安装系统的情况下，此时也具有 DHCP Client 及 TFTP Client 的能力。PXE

Client 发出请求到 DHCP Server 取得 IP 位址。然后, PXE Client 通过 TFTP 来下载 kernel 和 image 等文件。由 setup.cfg 文件可知 PXE-IPMI 驱动的代码入口为 ironic.drivers.pxe:PXEAndIPMIToolDrive:

```
class PXEAndIPMIToolDriver(base.BaseDriver):
    def __init__(self):
        self.power = ipmitool.IPMIPower()
        self.console = ipmitool.IPMIShellinaboxConsole()
        self.boot = pxe.PXEBoot()
        self.deploy = iscsi_deploy.ISCSIDeploy()
        self.management = ipmitool.IPMIManagement()
        self.inspect = inspector.Inspector.create_if_enabled(
            'PXEAndIPMIToolDriver')
        self.iscsi_vendor = iscsi_deploy.VendorPassthru()
        self.ipmi_vendor = ipmitool.VendorPassthru()
        self.mapping = {'send_raw': self.ipmi_vendor,
                        'bmc_reset': self.ipmi_vendor,
                        'heartbeat': self.iscsi_vendor,
                        'pass_deploy_info': self.iscsi_vendor,
                        'pass_bootloader_install_info': self.iscsi_vendor}
        self.driver_passthru_mapping = {'lookup': self.iscsi_vendor}
        self.vendor = utils.MixinVendorInterface(
            self.mapping,
            driver_passthru_mapping=self.driver_passthru_mapping)
        self.raid = agent.AgentRAID()
```

可以看出该驱动通过 IPMI 来进行电源管理, 采用 PXE 方式来启动, 采用 iSCSI 来部署目标镜像。该部署方式会通过 iSCSI 协议把硬盘挂载到部署服务器上, 然后把镜像部署到该硬盘之上, 使用 IPMIManagement 来设置启动方式和设备。raid 设置目前 Ironic 只支持 agent 方式。

Ironic 所在的节点就是部署服务器, 它既是 DHCP Server 也是 PXE server。Ironic 先把两次 PXE 启动所需要的两组 kernel+ramdisk 放置到 TFTP 服务目录下, 再通过 IPMI 远程启动目标机, 目标机启动之后, 发出 DHCP 请求, 从而开始第一次 PXE 启动。

在第一次 PXE 启动中, 传送到目标机的为 deploy_ramdisk 和 deploy_kernel。deploy_ramdisk 中内嵌了一个部署脚本, 该脚本会搜索目标机的硬盘, 并通过 iSCSI 把硬盘挂载到部署服务器上, 然后就阻塞并等待部署服务器的信号。这时部署服务器就会把之前已经准备好的镜像转换成 raw 格式并通过 dd 命令复制到目标机的硬盘上, 然后给目标机传送信号告诉它已经完成了复制。目标机接到信号后就会自动重启。

目标机重启之后就开始了第二次 PXE 启动。在这次 PXE 启动中, 传送到目标机的为 ramdisk 和 kernel, 与第一次 PXE 启动时并不相同, 它们通常又被称为 boot_kernel 和 boot_ramdisk。由于在第一次 PXE 启动中复制系统的时候, 只是 dd 了文件系统, 并没有

bootloader，这就意味着每次重启之后都需要通过 PXE 启动（PXE 启动本身当作 bootloader 来使用）来载入 kernel 才能正常启动。

在 Kilo 版本之后，Ironic 服务也支持本地启动，能够从本地的 bootloader 启动，这要求部署的 image 里包含 grub2 这个 bootloader。

在和 Nova Compute 服务一起工作的模式下支持本地启动需要设置 node 上的 capabilities:

```
$ ironic node-update <node-uuid> add
properties/capabilities="boot_option:local"
```

在 standalone 模式下这个参数需要设置在其他的地方，因为以上的 node capabilities 仅仅提供给 Nova Scheduler，而 standalone 模式不调用 Nova Scheduler:

```
$ ironic node-update <node-uuid> add
instance_info/capabilities='{ "boot_option": "local" }'
```

Kilo 之后的版本也已经支持部署到指定磁盘上:

```
$ ironic node-update <node-uuid> add properties/root_device='{ "wwn":
"0x4000cca77fc4dbal" }'
```

10.1.2 Ironic API

Ironic API 源码位于 ironic/api 目录:

```
.
├── acl.py
├── app.py
├── app.wsgi
├── config.py
├── controllers
│   ├── base.py
│   ├── link.py
│   ├── root.py
│   └── v1
│       ├── chassis.py
│       ├── collection.py
│       ├── driver.py
│       ├── node.py
│       ├── port.py
│       ├── state.py
│       ├── types.py
│       ├── utils.py
│       └── versions.py
├── expose.py
├── hooks.py
└── middleware
```


Ironic 使用 pecan 模块实现了 RESTful API 接口,使用 WSME 库来处理请求和返回对象,资源操作的实现都在 controllers 目录下面,主要的资源有 Chassis、Driver、Node、Port。

- Chassis: 表示一个机架或者机柜,它是 node 的集合。
- Driver: 表示服务里面的各种驱动资源,包括安装、部署、启动和电源控制。
- Node: 表示注册在 Ironic 中的一个物理机器。
- Port: 表示 Neutron 中网络端口,用来绑定物理机器上的网卡端口。

Node 资源是最核心的,其他资源被包含在 Node 中为其服务。

需要注意的是,Node 和 Driver 里面的 Vendor Passthru 子资源,它可以给某一 Node 或者 Driver 添加特定的函数,并可以在 Ironic API 中操作和调用这些 Vendor Passthru 函数,但 Ironic API 不会去解释和验证它们。

Node 里面的 Vendor Passthru 函数只对该 Node 有效,不区分 Driver,比如:

```
GET http://<address>:<port>/v1/drivers/pxe_ipmitool/vendor_passthru/
authentication_types
```

Driver 里面的 Vendor Passthru 函数只对该 Driver 有效,不区分 Node,比如:

```
POST {'raw_bytes': '0x01 0x02'} http://<address>:<port>/v1/nodes/<node
UUID>/vendor_passthru/send_raw
```

从 Kilo 版本开始, Ironic 支持 versions API, API 的版本号表示为 X.Y, 其中 X 为主版本号, Y 为次版本号, 目前主版本都为 1, Server 端在请求返回时指定它支持的最小次版本号 and 最大次版本号, 客户端可以在请求的 X-OpenStack-Ironic-API-Version 字段中指定请求的 API 版本号。

10.1.3 Ironic Conductor

从 Ironic API 过来的操作请求都会通过 RPC 调用 Conductor 的方法完成实际的工作,而 Conductor 又会根据配置使用具体的驱动,比如使用 PXE 去完成真正的部署。

Ironic Conductor 相关源码位于 ironic/conductor 目录:

```
.
├── base_manager.py
├── manager.py
├── notification_utils.py
├── rpcapi.py
├── task_manager.py
└── utils.py
```

Conductor 上的很多操作如耗时长的操作或者不能与其他并行进行的操作都需要在 ironic.conductor.task_manager.TaskManager 中完成, TaskManager 首先获取该操作节点的锁,

查询节点的 `node_id` 来获取该 `node` 的资源对象，如 `node`、`port`、`drivers`，启动一个单独的线程来工作，并在操作完成后释放该节点的锁。

下面以 `deploy` 为例进行说明，API 过来的部署请求由 `ironic.conductor.manager.ConductorManager` 类中的 `do_node_deploy` 函数来处理：

```
def do_node_deploy(self, context, node_id, rebuild=False,
                   configdrive=None):
    with task_manager.acquire(context, node_id, shared=False,
                             purpose='node deployment') as task:
        node = task.node
```

首先就是要通过 `TaskManager` 来获取该节点的锁，并在该锁的 `context` 里面完成所有的操作。接着验证驱动是否正确地包含操作所需要的信息：

```
try:
    task.driver.power.validate(task)
    task.driver.deploy.validate(task)
```

之后的所有操作都在 `TaskManager` 类 `process_event` 函数中：

```
task.process_event(
    event,
    callback=self._spawn_worker,
    call_args=(do_node_deploy, task, self.conductor.id,
               configdrive),
    err_handler=utils.provisioning_error_handler)
```

这个函数会在 `_spawn_worker()` 中启动一个线程来处理函数 `ironic.conductor.manager.do_node_deploy()`：

```
def do_node_deploy(task, conductor_id, configdrive=None):
    task.driver.deploy.prepare(task)
    new_state = task.driver.deploy.deploy(task)
```

假如这里是 `PXE_IPMI` 驱动，那么它会调用 `ISCSIDeploy` 的 `prepare` 函数来做部署准备工作，接着调用其 `deploy` 函数来部署。

用户可以同时启动多个 `Conductor` 服务，它们之间会协调地管理所有 `Ironic` 节点，把节点哈希到某一 `Conductor`，前提是该 `Conductor` 支持该节点的驱动。但这会带来一个问题，在可用的 `Conductor` 发生变化时需要重新哈希，导致节点与 `Conductor` 的对应关系发生很大的变动，这就可能需要为节点拆除或者重新搭建 TFTP，由此会导致 `Conductor` 服务的暂时不可用。

10.1.4 Ironic-python-agent

原本 `deploy_ramdisk` 只是完成了 `iSCSI` 部署的工作，但开发者觉得既然已经把 `kernel` 和 `ramdisk` 传过去了，只做这一个是太少了，而且还太缺乏灵活性了，所以就想在 `ramdisk`

里装一个 Python Agent。实际上就是多提供了一个 Restful API, 控制节点可以通过这个 Agent, 远程实现与物理结点的互动, 而不再仅仅是 dd 了。

这里是现有的部分 Use Case:

- 清除硬盘。
- 对硬盘分区。
- 安装 bootloader。
- 安装镜像。
- 升级固件。

10.1.5 ironic-inspector

Ironic-inspector 用来收集物理机器的各种硬件信息。和 Ironic-python-agent 类似, 它同样需要事先注入部署镜像中。它采用 RESTful Server 和 Client 的方式进行服务和通信。

Node 在被注册之后, 其状态被设置成 inspect 之后会执行 inspect 操作。ironic-inspector 采用非常灵活的插件机制来定义各种探测处理, 默认的探测比如 node 的 capacity、是否支持安全启动、CPU 虚拟化是否打开、节点上 GPU 的数量、网络接口的验证等。除了默认的探测处理还可以传入 JSON 格式的自定义的处理操作。

在 Mitaka 版本中还支持了节点自动发现, 如果 inspector 收到来自没有注册的节点的信息, 它会自动注册该节点。在 inspect 完成之后, node 的状态会变成 available, 这时可以进行部署了。

10.2 Ironic 中的网络管理

10.2.1 物理交换机管理

Nova 里面虚拟机的网络是在虚拟网络上构建的, 包括网卡、交换机、网卡与交互机之间的连级以及 L2 网络的联通都是虚拟的。在 Ironic 中物理机器间的网络都是在物理网络设备上构建的, 这就需要使得 Neutron 能够操作和配置物理交换机。为此有一个专门的 Neutron ML2 插件 Genericswitch 来做这个事情, Ironic 网络层面的逻辑架构如下:

```
OpenStack Neutron v2.0 => ML2 plugin => Generic Mechanism Driver => Device plugin
```

其中, Device plugin 使用 Netmiko 和 paramiko 库通过 ssh 来访问和配置物理交换机:

```
Device plugin => Netmiko => paramiko => ssh to switch
```

要使用这个插件, 首先要在 Neutron 的配置文件 etc/neutron/plugins/ml2/ml2_conf.ini 中添加 genericswitch 这个 driver:

```
[ml2]
tenant_network_types = vlan
type_drivers = local,flat,vlan,gre,vxlan
mechanism_drivers = openvswitch,genericswitch
.....
```

然后在文件/etc/neutron/plugins/ml2/ml2_conf_genericswitch.ini 中配置具体的交换机 driver 信息:

```
[genericswitch:<switch name>]
device_type = <netmiko device type>
ip = <IP address of switch>
port = <ssh port>
username = <credential username>
password = <credential password>
key_file = <ssh key file>
secret = <enable secret>
```

最后启动或者重启 neutron-server:

```
$ neutron-server \ --config-file /etc/neutron/neutron.conf \ --config-file
/etc/neutron/plugins/ml2/ml2_conf.ini \ --config-file
/etc/neutron/plugins/ml2/ml2_conf_genericswitch.ini
```

目前该插件支持的交换机有:

- Cisco IOS switches;
- Huawei switches;
- OpenVSwitch;
- Arista EOS。

10.2.2 多租户网络的支持

在实现 Ironic 多租户网络的支持之前, Ironic 只支持 Flat 网络模式, 也就是所有的节点都在同一个 L2 网络之中。利用 Neutron 实现多租户之间的网络隔离, Ironic 可以提供以下 3 种类型的网络。

- Noop: 给 standalone 使用, 不对网络做任何操作。
- Flat: 所有 Node 放置在一个 L2 网络中。
- Neutron: 多租户网络, 租户之间实现了网络隔离。

Ironic 利用链路层发现协议 (Link Layer Discovery Protocol) 获取物理网络连接信息, 并把这些 local_link_connection 信息发送给 Neutron ML2 plugin, 以便 driver 在 provision 服务器的时候配置与节点连接的机架交换机 (TOR, the top-of-rack) 上的端口。

这些 local_link_connection 信息包含以下方面的内容。

- `switch_id`: 标识一个交换机, 可以是一个 MAC 地址或者一个基于 OpenFlow 协议的软交换机。
- `port_id`: 交换机上的端口号, 比如 Gig0/1。
- `switch_info`: 可选项, 用来区分不同的交换机型号或者 `vendor` 特有的标识。

在 Neutron 多租户网络中, `provisioning` 网络和 `cleaning` 网络必须在同一或者不同的某一指定的 Neutron L2 层网络之中, 这时所有的 `node` 都处于同一个 L2 网络之中, 这是由于比如在 PXE 启动过程中需要连接 TFTP 服务器, 而 TFTP 服务器并不与租户网络联通。如果需要实现在 `provisioning` 和 `cleaning` 阶段的租户网络隔离, 则要求每个租户网络都有各自的 TFTP 服务器或者全局 TFTP 服务器通过路由连接租户网络, 目前在 `Ironic` 中并不支持该功能。同样的原因, 配置 Neutron 多租户网络必须开启本地启动功能, 否则每次启动都需要在 PXE 启动过程中连接 TFTP 服务器。

要配置多租户网络, 首先配置 `Ironic` 服务, 具体代码如下:

```
enabled_network_interfaces=noop,flat,neutron
# 需要配置在 conductor node 上, 同时要在 API 配置文件中设置
default_network_interface=neutron
cleaning_network_uuid=$CLEAN_UUID
# 需要配置在 conductor node 上
provisioning_network_uuid=$PROVISION_UUID
# 需要配置在 conductor node 上
# 这 2 个网络可以是同一个 Neutron 网络, 最好 security group 是关闭的, 否则需要保证
# DHCP/PXE/TFTP 等一系列服务端口打开
Restart the ironic conductor and API services
# 重启 Conductor 和 API 服务
```

然后, 配置 `Ironic Node` 节点, 具体代码如下:

```
enabled_network_interfaces=noop,flat,neutron
# 需要配置在 conductor node 上, 同时要在 API 配置文件中设置
default_network_interface=neutron
cleaning_network_uuid=$CLEAN_UUID //on conductor
provisioning_network_uuid=$PROVISION_UUID //on conductor
# 这 2 个网络可以是同一个 Neutron 网络, 最好 security group 是关闭的。
Restart the ironic conductor and API services
```

最后重启 `Ironic Conductor` 和 `API` 服务。

10.3 Ironic 节点的注册和启动

首先是注册一个节点, 在注册节点的时候需要指定该节点的 `driver`, 具体代码如下:

```
$ ironic node-create -d pxe-ipmitool
```

Property	Value
uuid	dfc6189f-ad83-4261-9bda-b27258eb1987
driver_info	{}
extra	{}
driver	pxe_ipmitool
chassis_uuid	
properties	{}
name	None

注册完成之后该节点的状态就变成 Available。

\$ ironic node-show dfc6189f-ad83-4261-9bda-b27258eb1987

Property	Value
chassis_uuid	
clean_step	{}
console_enabled	False
created_at	2016-10-14T01:45:50+00:00
driver	pxe_ipmi
driver_info	{}
driver_internal_info	{}
extra	{}
inspection_finished_at	None
inspection_started_at	None
instance_info	{}
instance_uuid	None
last_error	None
maintenance	False
maintenance_reason	None
name	None
power_state	None
properties	{}
provision_state	available
provision_updated_at	None
raid_config	
reservation	None
target_power_state	None
target_provision_state	None
target_raid_config	
updated_at	None
uuid	dfc6189f-ad83-4261-9bda-b27258eb1987

填充或者更新 `driver_info` 使得 `ironic` 服务知道如何来管理节点，比如：

```
$ ironic driver-properties pxe_ipmitool ironic node-update $NODE_UUID add \
\ driver_info/ipmi_username=$USER \
driver_info/ipmi_password=$PASS \
driver_info/ipmi_address=$ADDRESS
```

根据之前定义的 `bare metal flavor` 来更新 `node` 的 `properties`，具体代码如下：

```
$ ironic node-update $NODE_UUID add \
properties/cpus=$CPU \
properties/memory_mb=$RAM_MB \
properties/local_gb=$DISK_GB \
properties/cpu_arch=$ARCH
```

设置 `hardware capabilities` 给 `scheduler`，具体代码如下：

```
$ ironic node-update $NODE_UUID add \
properties/capabilities=key1:val1,key2:val2
```

为 `node driver` 指定 `deploy kernel and ramdisk`：

```
$ ironic node-update $NODE_UUID add \
driver_info/deploy_kernel=$DEPLOY_VMLINUZ_UUID \
driver_info/deploy_ramdisk=$DEPLOY_INITRD_UUID
```

为 `node` 创建 `port`，需要该 `node` 的网卡 `MAC` 地址以便传给 `Neutron` 服务，具体代码如下：

```
$ ironic port-create -n $NODE_UUID -a $MAC_ADDRESS
```

最后验证 `ironic` 是否具备必要的信息来启动 `node driver`，具体代码如下：

```
$ ironic node-validate $NODE_UUID
+-----+-----+-----+
| Interface | Result | Reason |
+-----+-----+-----+
| console   | True   |         |
| deploy    | True   |         |
| management | True   |         |
| power     | True   |         |
+-----+-----+-----+
```

节点注册好了之后，通过 `Nova boot API` 调用 `Ironic` 来完成部署。

1) 首先要在 `nova` 服务中配置 `Ironic driver`，具体代码如下：

```
$ ironic node-show $NODE
[default]
compute_driver=ironic.IronicDriver
scheduler_host_manager=ironic_host_manager
ram_allocation_ratio=1.0
```

```
reserved_host_memory_mb=0
```

2) 通过 nova boot 启动, 和普通 nova boot 唯一的区别在于这里需要定义特殊的 flavor, extra_specs 里面需要包含之前已经定义好的 deploy_kernel/deploy_ramdisk:

```
"baremetal:deploy_kernel_id": "597ebb83-ffc5-4ff7-910c-3c7e953ea121"  
"baremetal:deploy_ramdisk_id": "6e8d87bb-cda9-4c21-9e94-b39a4ca995f5"
```

3) 接着, 我们就可以看到 Ironic 开始部署了:

通过 Scheduler 选取合适的物理节点, 通过 IPMI 启动物理节点, 并向 DHCP 服务器申请 IP。

这时候开始了第一次 PXE 启动, 通过网络传输 deploy_kernel/deploy_ramdisk 给物理节点。ramdisk 已经内嵌了脚本, 通过 iSCSI 挂载物理节点上的硬盘到控制节点, 完成镜像复制, 并通知控制节点。

然后 IPMI 重启物理节点, 再次通过 PXE 启动, 并读取硬盘, 切换到硬盘上的系统, 完成部署。

在用户能够登录目标机之前还有一个很重要的步骤就是初始化, 这通过 Cloud-init 来实现。Cloud-init 原本就是用来完成云环境中虚拟机实例的初始化的, 已经应用在亚马逊的 EC2 虚拟机实例的初始化中, 其作用主要包括以下几个:

- 设置实例的主机名;
- 生成实例的 SSH 的私钥;
- 添加用户的 SSH 密钥到实例, 从而便于用户登录。

控制面板

Openstack 需要提供一个简洁方便、用户友好的控制界面给最终的用户和开发者，让他们能够浏览并操作属于自己的计算资源，这就是 Openstack 的控制面板(Dashboard)项目 Horizon。

早期的 Horizon 只是一个简单的 App，仅操作和处理 Openstack Compute 相关的项目。后来随着其他项目的迅速加入，对原来的简单视图和 API 的调用提出了更高的要求。时至今日，Horizon 已具备或致力于具备的核心价值发展为：

- 1) 核心支持——支持所有的 Openstack 项目；
- 2) 可扩展性——每个开发者都能增加组件；
- 3) 易于管理——架构和代码易于管理，浏览方便；
- 4) 视图一致——各组件的界面和交互模式保持一致；
- 5) 可兼容性——API 向后兼容；
- 6) 易于使用——界面用户友好。

11.1 Horizon 体系结构

Horizon 提供了一个模块化的基于 Web 的图形界面，用户可以通过浏览器使用 Horizon 提供的控制面板来访问和控制它们的计算、存储和网络资源，比如启动虚拟机实例、创建子网、分配 IP 地址、设置访问控制等。

Horizon 采用了 Django 框架，简单地说，它就是个单纯的基于 Django 的网站，Horizon 同时采用了许多流行的前端技术扩展其功能，如 Bootstrap、jQuery、Underscore.js、AngularJS、D3.js、Rickshaw 和 LESS CSS 等。

11.1.1 Horizon 与 Django

Django 是一种流行的基于 Python 语言的开源 Web 应用程序框架，Horizon 遵循 Django 框架的模式生成若干 App，合在一起为 OpenStack 控制面板提供完整的实现。

Django 是由美国堪萨斯州 Lawrence 市的一个新闻网站的开发小组开发出来的，旨在以最小的代价构建和维护高质量的 Web 站点。在著名的《The Django book》一书中，详细地描述了它所紧密遵循的设计哲学——MVC 设计模式。

在 Django App 中，一般有 4 种文件存在，分别是 models.py、views.py、urls.py 以及 html 网页文件。

```

# models.py (database tables)
from django.db import models

class Music(models.Model):
    name = models.CharField()
    producer = models.CharField()
    pub_date = models.DateField()

# views.py (business logic)
from django.shortcuts import render_to_response
from models import Music

def latest_music(request):
    music_list = Music.objects.order_by('-pub_date')[:10]
    return render_to_response('latest_music.html', {'music_list':
music_list})

# urls.py (URL configuration part)
from django.conf.urls.defaults import *
import views

urlpatterns = patterns('',
    (r'^latest/$', views.latest_music),
)

# latest_music.html (the template)
<html><head><title>Music</title></head>
<body>
<h1>Music</h1>
<ul>
{% for music in music_list %}
<li>{{ music.name }}</li>
{% endfor %}
</ul>
</body></html>

```

1) `models.py` 使用 Python 类来描述数据表及其数据库操作，称为模型 (Model)。上述示例中 `models.py` 文件主要用一个 Python 类来描述数据表。在这个类中通过 Python 的代码替代 SQL 语句来创建、更新、删除、查询数据库中的记录。

2) `views.py` 包含页面的业务逻辑 (Business Logic)，该文件里的函数通常叫做视图 (View)。

3) `urls.py` 描述了当浏览器网址指向哪一级的目录时,Python 解释器需要调用哪个视图去渲染网页。上述示例中`/latest/URL` 将会调用 `latest_music()`函数生成相应的视图。

4) `latest_music.html` 网页文件则主要负责网页设计,一般内嵌模板语言以实现网页设计的灵活性。

这 4 种文件以这种松散耦合的方式组成的模式是模型/视图/控制器 (MVC) 的一个基本范例。MVC 模型的定义如下所述。

- M: 数据存取部分, 由 Django 数据库层处理。
- V: 选择显示哪些数据, 以及怎样显示, 由视图和模板处理。
- C: 根据用户输入选择视图的部分, 由 Django 框架根据 `URLConf` (URL 配置) 设置, 对给定 URL 调用适当的 Python 函数。

由于 Django 框架自动处理 C 的部分, Django 里更关注的是模型、模板 (Template) 和视图, 也因此又被称为 MTV 框架。

- 1) M: 代表模型 (Model), 即数据存取层。
- 2) T: 代表模板 (Template), 即表现层。
- 3) V: 代表视图 (View), 即业务逻辑。

在 MTV 的框架模式中, Django 视图不处理用户输入, 而仅仅决定要给用户展现哪些数据; Django 模板仅仅决定如何展现 Django 视图指定的数据。换句话说, Django 将 MVC 中的视图进一步分解为 Django 视图和 Django 模板两个部分, 分别决定“展现哪些数据”和“如何展现”, 使得 Django 的模板可以根据需要随时替换, 而不仅仅限制于内置的模板。

Django 框架的显著优点是 App 的每个部分各行其是, 部分的改动不影响其他。比如决定哪个 URL 网址目录调用哪个视图是在 `urls.py` 中实现的, 而该视图是如何实现的则在 `views.py` 中, 所以这两个文件的开发工作可以互不干扰。

把这 4 种文件分开, 前面 3 种用 Python 代码实现, HTML 网页则让网页设计人员完成, 这种方式自然令代码更简洁、高效。

Django 提供了模板系统 (Template System) 来进一步实现这个目标。模板是定义了占位符和模板标签 (Template Tag) 的文本, 它将网页的显示形式和内容分离开, 并且用占位符和模板标签提供各种逻辑去规范网页的显示。在用 Django 开发的 App 的目录下, 我们可以看到名为 `templates` 和 `templatetags` 的目录, 它们是用来处理模板和模板标签的。

在 Django 的设计哲学中, 业务逻辑是与表现逻辑分开的。Django 的视图是代表业务逻辑的, 模板系统则被视作控制表现和与表现逻辑相关的工具。通常的用法是: 视图向模板传入上下文 (Context), 模板利用传入的上下文渲染网页。所以视图负责得到上下文部分, 模板负责显示。这就是 Django 作为 Web 应用程序框架的基本哲学。

Django 遵循 DRY (Don't Repeat Yourself) 原则, 专注于代码的高度可重用, Horizon 秉承了这种哲学, 致力于支持可扩展的控制面板的框架, 尽可能重复利用已有的模板开发和管理的 OpenStack 网站。

基于 Django, Horizon 体系结构如图 11-1 所示。底层 API 模块 `openstack_dashboard.api` 将 OpenStack 其他项目的 API 封装起来供 Horizon 其他模块调用。需要注意的是, 该 API 模块只提供了其他项目 API 的一个子集。Horizon 将页面上的所有元素模块化, 表单、表格等一些网页中常见的元素全部被封装成 Python 类, 如图 11-1 中的 View 模块所示。每个这样的组件都有自己对应的一块 HTML 模板, 渲染整个页面时, Horizon 会先找到当前页面包含多少组件, 并将各个组件分别渲染成一段 HTML 片段, 最后拼接成一个完整的 HTML 页面返回给浏览器。

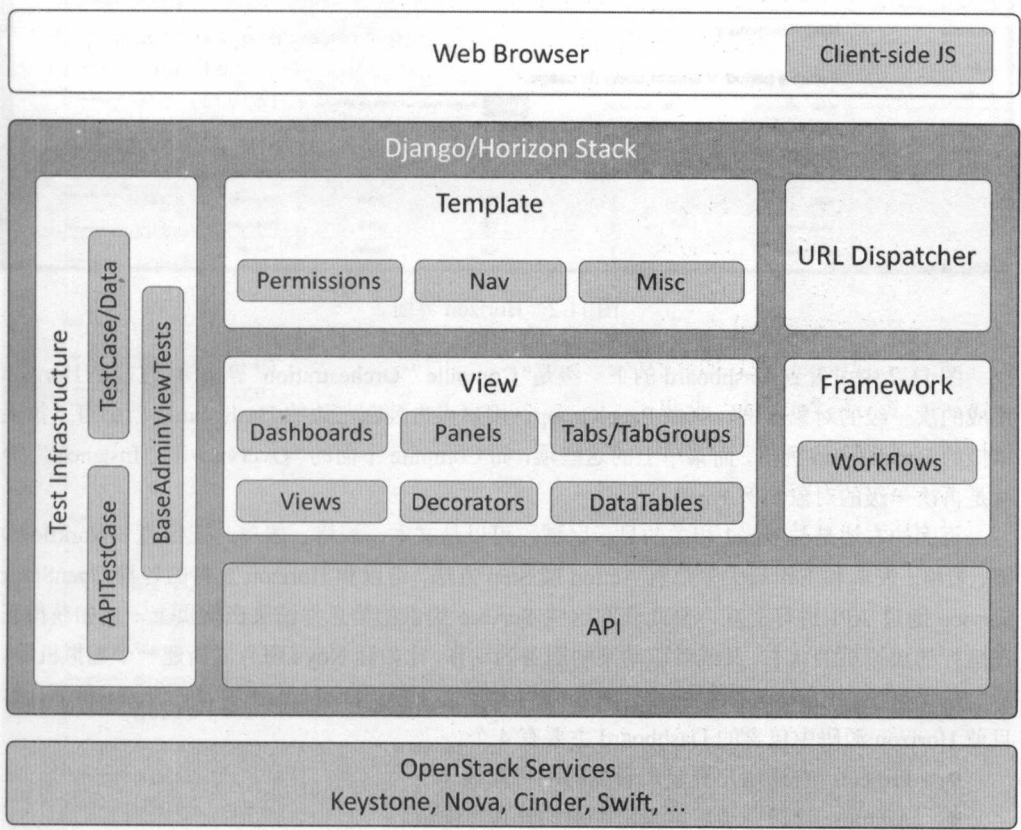


图 11-1 Horizon 体系结构

11.1.2 Horizon 网站布局

图 11-2 所示为一个由 Horizon 创建和维护的网站主页。页面的左边, “Project” “Identity” 等按键是 Horizon 项目生成的与显示有关的最上层的对象实例, 它们都是 OpenStack 网页的一个 Dashboard (仪表盘)。

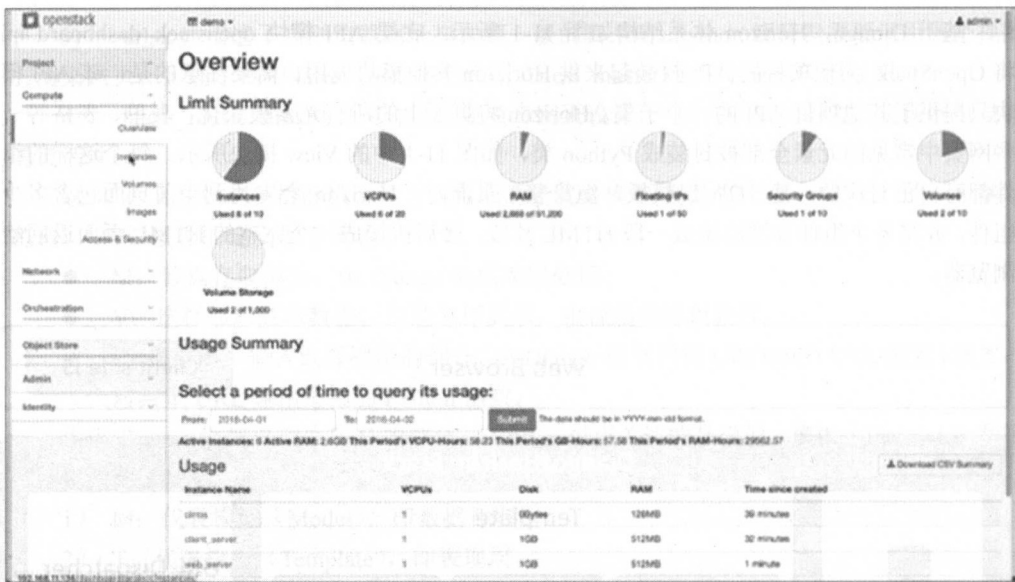


图 11-2 Horizon 界面

图 11-2 中可看到 Dashboard 的下一级是“Compute”“Orchestration”等按键,这些是 Horizon 生成的次一级的对象实例,叫做 PanelGroup。用户点击列在左边的 Dashboard,就会有下拉菜单似的 PanelGroup 列出,而菜单上的这些项,如 Compute 下面的“Overview”“Instances”等就是再次一级的对象实例 Panel。

页面的右边是与 panel 相关的显示区域,可以是文本、表格、表单、工作流 (Workflow) 等。表格、表单和工作流大多都有 Action 或 Step 方法,可以和 Horizon 支持的各种 OpenStack Service 通过 API 进行交互,借此获取这些 Service 提供的信息并渲染在网页上,比如获得正在运行的虚拟机的状态,又或者驱动某些服务的动作,比如让 Nova 服务器新建一个虚拟机等。

总结来说,Horizon 面板的设计就是分为上述的 3 层: Dashboard、PanelGroup 和 Panel。目前 Horizon 源码中包含的 Dashboard 主要有 4 个。

- Project: 普通用户登录后看到的项目面板。
- Admin: 管理登录后可见,左侧的管理员面板。
- Settings: 右上角的设置面板,里面可设置语言、时区、更改密码。
- Router: 配置文件中将 profile_support 打开可见, Cisco nexus 1000v 的管理面板。

Horizon 除了提供这些主要的网页界面功能外,还通过配置文件的各种选项控制网页的其他细节。比如通过配置文件 local_settings.py 设置 OpenStack 主页上的 Logo 图片,或者指定这个主页的标题,比如图 11-1 页面最上方显示的“Openstack Dashboard”字符串。

11.1.3 Horzion 源码结构

Horizon 源码可以从 <https://github.com/openstack/horizon.git> 获取，在源码根目录下有两个主要的目录，分别是 `horizon` 和 `openstack_dashboard`，它们也是 Horizon 项目的两个主要的组成部分。`horizon` 目录结构如下：

```
.
├── base.py
├── browsers
├── conf
├── context_processors.py
├── contrib
├── decorators.py
├── exceptions.py
├── forms
├── hacking
├── karma.conf.js
├── loaders.py
├── locale
├── management
├── messages.py
├── middleware
├── notifications.py
├── site_urls.py
├── static
├── tables
├── tabs
├── templates
├── templatetags
├── test
├── themes.py
├── utils
├── version.py
├── views.py
├── views.pyc
└── workflows
```

`base.py` 该文件定义了 `horizon` 模块中从 `HorizonSite` 到 `Panel` 的各种组件的类库，并且用类 `HorizonSite` 为整个项目实例化了一个 `horizon` 对象。在 `HorizonSite` 之下的类依次是 `Dashboard` 类、`PanelGroup` 类和 `Panel` 类。

`browsers`（浏览器）、`forms`（表单）、`tables`（表格）、`tabs`（标签）和 `workflows`（工作流）等目录分别对应着不同的视图类实现。这些视图类继承了 Django 提供的通用视图（Django 的 `Generic` 库），既很好地利用了 Django 框架的可扩展性，又添加了新的视图特性，为网页的开

发提供了更为精细灵活的实现。当用户单击图 11-2 中的某个 Panel 时，右边的页面上就是由这些视图类渲染的。

template 和 templatetags 目录是 Django 框架加载网页时用到的模板和模板标签。

horizon 模块的实现充分利用了很多 Django 框架提供的一些高级特性。比如文件 context_processors.py 是一个自动设置相应上下文变量来解析模板的上下文处理器，它把 HORIZON_CONFIG 字典自动加到了 Context 上下文变量中传给模板。

loaders.py 帮助 horizon 模块使用自定义的方式加载模板而非使用 Django 自带的模板加载器。

middleware.py 提供 horizon 中间件，用来处理收到异常和网页请求及回应需要附加的动作。

decorators.py 用于方便取得网页请求的当前组件和权限认证。

horizon 目录之外，另一个主要目录是 openstack_dashboard。如果把 horizon 目录中的各种实现当成积木，那么搭好的小房屋就放在 openstack_dashboard 的目录下。目录 openstack_dashboard 的源码结构如下：

```
.
├── api
├── conf
├── context_processors.py
├── contrib
├── dashboards
├── django_pyscss_fix
├── enabled
├── exceptions.py
├── hooks.py
├── karma.conf.js
├── local
├── locale
├── management
├── policy.py
├── settings.py
├── static
├── static_settings.py
├── templates
├── templatetags
├── test
├── themes
├── theme_settings.py
├── urls.py
├── usage
├── utils
├── views.py
└── wsgi
```

在 `openstack_dashboard` 目录下，有一个重要的项目配置文件 `settings.py`，它是每个应用 Django 框架的项目必需的配置文件，通常用它来指定项目部署的重要文件路径和方式。

目录 `dashboards` 下的每一个子目录都是项目的一个 Dashboard。每一个 Dashboard 都有一个 `dashboard.py` 来声明。该文件把在 `horizon/base.py` 文件中声明的 Dashboard 类和下属的各种 PanelGroup 类实例化，并且将该 Dashboard 实例注册在全局的 `horizon` 实例中。前面我们提到过，`horizon` 实例是在 `base.py` 文件中声明过的 `HorizonSite` 的类实例。

每个 Dashboard 可以拥有多组的 PanelGroup。而对每一个 PanelGroup 的实例化则声明了该 PanelGroup 下属的各个 Panel。Horizon 实行层层注册制度，每个 Dashboard 实例注册在全局的 `horizon` 实例中，每个它下属的 PanelGroup 注册在该 Dashboard 中，而每个 Panel 则注册在它所属的 PanelGroup 里。每一个 Panel 的名字都有对应的子目录。子目录的 `panel.py` 文件则实例化了对应的 Panel。Panel 类的声明也在 `horizon/base.py` 中。

在网页的实现上，单击 Panel 会触发网页主要区域的渲染，这可能有文本、表格、表单、标签或者工作流等，所以在代表 Panel 的目录下，可以看到相应的 `tables.py`、`forms.py`、`tabs.py` 和 `workflows` 等文件或目录。这些文件包含该 Panel 渲染的视图实体类的实现，它们通常要用到相应的 `horizon` 子目录下对应的实现。比如 `tables.py` 文件中用到的父类就在 `horizon/tables` 目录下的文件中，`forms.py` 文件中用的父类则在 `horizon/forms` 目录下的文件中。代表 Panel 的目录下有我们熟悉的 `urls.py` 文件、`views.py` 文件和 `template` 目录等 Django 框架的基本文件。`views.py` 负责渲染每块需要动态渲染区域。`urls.py` 负责选取哪个视图去渲染哪块区域。

另一个重要的目录是 `api`。这个目录下的文件是网站提供的 OpenStack 各项服务与 Horizon 的接口，多数以项目服务名为文件名。比如与 Nova 服务的接口在 `nova.py` 中。这些文件请求它所代表的服务的 client，包装这些 client 提供的各种接口方法给 `openstack_dashboard` 使用。各个 Dashboard 下的 Panel 们使用各种视图文件显式地调用这些方法，从而让用户在网页上使用各种服务和浏览服务信息。

11.2 Horizon 部署

Horizon 既可以单独安装，也可以通过 Devstack 和其他 OpenStack 项目一起安装。通过 Devstack 安装时，不必再费心配置其他的 OpenStack 服务与 Horizon 配合，可以装好即用，但我们可以通过对 `openstack_dashboard/settings.py` 文件进行修改来更改一些配置。

1. 配置

`openstack_dashboard/settings.py` 是 Horizon 的主要配置文件。在这个配置文件中主要有 3 个方面的内容。

(1) 有关 Django 框架的基本设置，用来规范 Django 的开发环境。

- `INSTALLED_APPS`：用于指定项目是由哪些 Django APP 组成的，这个字典里除了

Django 自带的一些通用的 APP，还有 `opstack_dashboard`、`horizon`、`openstack_auth` 等 APP，这些 APP 组合起来实现了网站。

- `ROOT_URLCONF`：用于指定这个网站所有网页的根 URL 是记录在文件 `openstack_dashboard/urls.py` 中的。
- `TEMPLATE_DIR`：用于指定项目模板所在的根目录。
- `MIDDLEWARE_CLASSES`：用于指定项目用到的中间件，包括 Django 自带的中间件和 `horizon` 实现的中间件。
- `TEMPLATE_CONTEXT_PROCESSORS`：用于指定项目要使用哪些模板上下文处理器。这里我们看到除了 Django 自带的，`horizon` 和 `openstack_dashboard` 都实现了自己开发的模板上下文处理器。
- `TEMPLATE_LOADERS`：用于指明项目要使用的模板加载器，除了 Django 自带的，`horizon` 实现了自己的模板加载器。
- `STATICFILES_DIRS`：用于指明项目中利用的其他网页设计框架包的所在路径。比如 `angularjs`、`jquery` 和 `bootstrap` 等。

(2) `HORIZON_CONFIG` 字典，字典里的选项用来规范项目的安装配置。

- 静态地指定默认的 Dashboard 和 Dashboard 的加载顺序，因与 Pluggable Settings（可插拔设置）冲突，不同时使用。Pluggable Settings 为新加入的特性用来动态地加载网页的 Dashboards、PanelGroups 和 Panels。如果有了 Pluggable 设置后，就不需要在这里静态指定。
- 含有这些 Pluggable 设置的文件放在两个目录下，分别是 `openstack_dashboard/enabled` 和 `openstack_dashboard/local/enabled`。目录下的每个文件指定该 Dashboard 是否加载。`Horizon` 在加载网页时，先寻找前面一个目录下的文件，按字母顺序加载，后一个目录里的文件可以覆盖前面目录文件的加载。这可以使得开发者可以在不改变网页默认布局的同时，方便地添加或改变 `Horizon` 组件。如同 Dashboard 一样，PanelGroup 和 Panel 也都可用同样方式轻松地添加或删除。
- 规范使用其他网页开发技术的细节。比如 AJAX (Asynchronous Javascript And XML)。AJAX 通过使用 Javascript 的 XMLHttpRequest 的对象，可以实现异步请求并支持更新部分网页内容而不重载整张网页。`Horizon` 指定了应用 AJAX 的一些限制，如连接数目和侦听频率等。再比如对 AngularJS，一种网页开发技术端对端的框架，指定包含和加载它的哪些模块。
- 与 `openstack_dashboard` 没有明确关联的一些其他配置选项。

(3) 引入 `openstack_dashboard/local/local_settings.py` 文件，借此定义了一些与 OpenStack 其他的项目或服务相关的设置，规范 `Horizon` 和其他 OpenStack 项目之间进行交互的 API 的细节。比如指定 Keystone 的 URL 来进行认证，又比如定义一个字典来指定 Neutron 提供的网络特性等。

2. 测试

源码根目录下的 `run_tests.sh` 文件用来验证整个 Horizon 项目 Python 代码的稳定性，通常只需在命令行下直接运行该文件即可。

```
$ ./run_tests.sh
```

或者是更完整的测试：

```
$ ./run_tests.sh -with-selenium
```

也可以是只测试 Horizon 源码目录中提供的各级 `tests` 目录下的文件。这些是开发者撰写的单元测试文件。比如：

```
$ ./run_tests.sh horizon.test.tests.base
```

我们还可以利用 `run_tests.sh` 文件方便地生成 Dashboards 和 Panels。`run_tests.sh` 的这个特点有点类似于一些 GUI 开发环境，通过一些预定义好的控件来组合成最后的图形界面，节省了手工去编写大量控件基本实现代码的时间。比如我们添加一个 Dashboard，可以不用首先去 `openstack_dashboard/dashboards` 目录下添加对应的子目录以及实现，而是可以执行下面的命令：

```
$ ./run_tests.sh -m startdash "your dashboard"
```

然后就会在 `openstack_dashboard/dashboards` 目录下生成新命名的 `dashboard` 目录，及简单的 `dashboard.py` 文件。

同样可以在新生成的 `dashboard` 目录下，生成新命名的 `Panel` 目录及简单的 `Panel` 文件。

```
$ ./run_tests.sh -m startpanel "your panel" --dashboard = "dashboard path"
```

`run_tests.sh` 的更多功能可参见 http://docs.openstack.org/developer/horizon/ref/run_tests.html。

11.3 页面渲染流程

图 11-2 所示的页面主要由各个 Dashboard 和 Panel 以及单击 Panel 后渲染的页面组成，其中，单击 Panel 请求的页面实现得最为复杂。各个页面的实现当然不尽相同，但还是有大致脉络可寻，本节主要讲述这部分页面渲染的一般流程。

如果登录进一个用 Devstack 搭建的 OpenStack 网站，一般默认显示的是一个“Instance Overview”的页面，如图 11-3 所示。可以看到，当前的 Dashboard 是“Project”，PanelGroup 是“Project”下的“Compute”，Panel 是“Compute”下的“Overview”。右边的区域就是单击请求“Overview”Panel 渲染的页面部分。

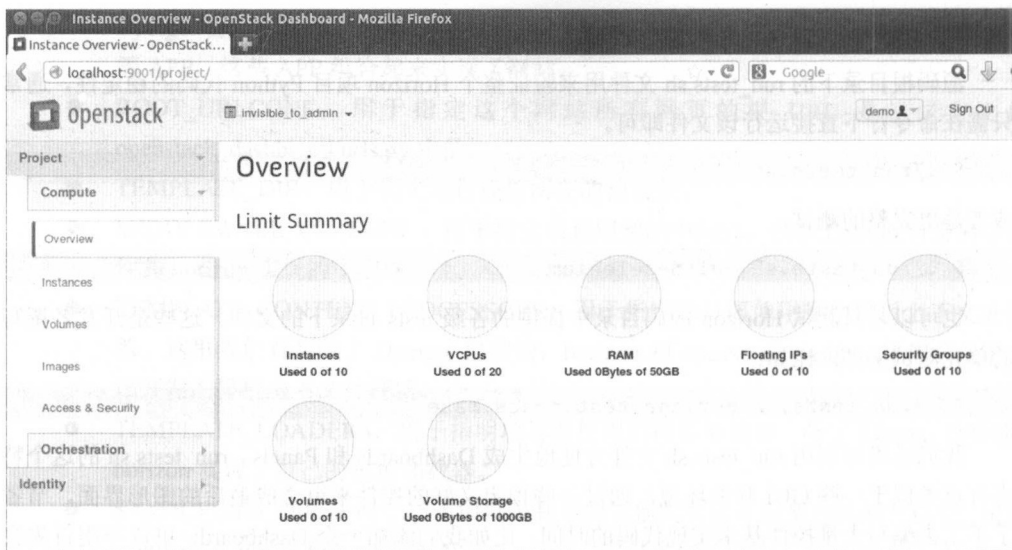


图 11-3 默认的“Instance Overview”页面

如前所述，所有 Dashboard 的实现都位于 `openstack_dashboard/dashboards` 目录中。其中，名为“project”的子目录就是“Project”Dashboard 的实现，里面的文件 `dashboard.py` 声明了 Project 这个 Dashboard，并把它注册在 horizon 的实例上，同时指明它包含的几个 PanelGroup。

```
class Project(horizon.Dashboard):
    name = _("Project")
    slug = "project"

    def can_access(self, context):
        request = context['request']
        has_project = request.user.token.project.get('id') is not None
        return super(Project, self).can_access(context) and has_project

horizon.register(Project)
```

该文件只是声明了 Dashboard 的名称，却并没有指明该 Dashboard 会包含哪些 PanelGroup 以及默认显示的 Panel。原因在于相比旧版本，Horizon 将所有的 PanelGroup 以及 Panel 的注册都统一通过 plugin 的方式进行，并将相关配置文件放在 `openstack_dashboard/enabled` 目录下，而 `openstack_dashboard/dashboards` 只专注于 panel 相关功能的实现。

如“Project”Dashboard 下属的“Compute”PanelGroup 的声明就在 `openstack_dashboard/enabled/_1010_compute_panel_group.py` 文件中，其通过 `PANEL_GROUP_DASHBOARD` 变量将“Compute”PanelGroup 连接于“Project”Dashboard。

```
# The slug of the panel group to be added to HORIZON_CONFIG. Required.
```

```
PANEL_GROUP = 'compute'
# The display name of the PANEL_GROUP. Required.
PANEL_GROUP_NAME = _('Compute')
# The slug of the dashboard the PANEL_GROUP associated with. Required.
PANEL_GROUP_DASHBOARD = 'project'
```

“Compute” PanelGroup 包含若干 Panel，参见图 11-3。以默认“overview” Panel 为例，它的实现位于下一级子目录 `openstack_dashboard/dashboards/project/overview`。该目录下的文件 `panel.py` 很简单，声明一个 `panel` 类 `Overview`，并将它注册到“Project”Dashboard 上。代码中的这些父类和注册函数的实现都在文件 `horizon/base.py` 中。

```
class Overview(horizon.Panel):
    name = _("Overview")
    slug = 'overview'
```

同理，“overview” Panel 的注册也是通过位于 `openstack_dashboard/enabled` 目录下的 `_1020_project_overview_panel.py` 实现的。该文件将“overview” Panel 链接于“Compute”PanelGroup，并设为默认的 Panel。

```
# The slug of the panel to be added to HORIZON_CONFIG. Required.
PANEL = 'overview'
# The slug of the dashboard the PANEL associated with. Required.
PANEL_DASHBOARD = 'project'
# The slug of the panel group the PANEL is associated with.
PANEL_GROUP = 'compute'

# If set, it will update the default panel of the PANEL_DASHBOARD.
DEFAULT_PANEL = 'overview'

# Python panel class of the PANEL to be added.
ADD_PANEL = 'openstack_dashboard.dashboards.project.overview.panel.Overview'
```

那么单击 Panel 请求的页面是怎么渲染的呢？按照 Django 的开发模式，页面的内容是用视图函数生成的，页面的 URL 则由 `URLConf` 来指定。在 `overview` 子目录下的 `urls.py` 文件，指明了 `URLConf`，即浏览器显示的地址中的目录使用 `overview` 模块中的哪个视图函数去渲染。

```
urlpatterns = patterns(
    'openstack_dashboard.dashboards.project.overview.views',
    url(r'^$', views.ProjectOverview.as_view(), name='index'),
    url(r'^warning$', views.WarningView.as_view(), name='warning'),
)
```

`patterns` 函数的第一个参数是用字符串显式地表示该页面所用的通用视图函数前缀，该字符串中的“.”可以就当作“/”解，它指出当前要用到的视图函数都在模块文件 `openstack_dashboard/dashboards/project/overview/views.py` 中。

`patterns` 函数中的 `url()` 参数以 Python 元组为参数，定义要显示的 URL 网页目录和对应视图函数的映射，这也叫做一个 URLpattern。该元组中的第一个参数用正则表达式匹配网页目录的字符串，第二个参数表示该页的内容由视图函数 `ProjectOverview.as_view()` 提供，第三个参数则是传给该视图函数使用的参数。这两个 URLpattern 表示单击“overview”Panel 收到的请求可能在网页浏览器中显示两个地址：一个是当前根目录，另一个是根目录下名为“warning”的子目录。跟进指定目录下的 `views.py` 文件去看看这两个视图函数的实现。

```
from horizon import views
```

```
class WarningView(views.HorizonTemplateView):  
    template_name = "project/_warning.html"
```

先看第二个 URLpattern 对应的视图函数，它相对简单，几乎没有包含任何 Horizon 的实现。类 `WarningView` 继承自 Django 的类 `TemplateView`。对 Horizon 来说，`TemplateView` 也可算是模板渲染机制的一个基类。可以参考 Django 包中源码，比如 `django/views/generic/base.py` 等文件，看看 Django 用模板实现视图的机制。

在 Django 框架中，视图就是一个 Python 函数，它接收 `HttpRequest` 的参数，返回 `HttpResponse` 对象。Django 得到这个返回对象后，将它转换成对应的 HTTP 的响应，显示网页内容。

为了代码的更高的可重复性，Django 实现了模板系统，把原始视图的参数和返回值及转换响应等实现都包装成了各种类和方法，`TemplateView` 类是其中之一，在 URLpattern 中出现的 `WarningView.as_view()` 等视图函数是 `TemplateView` 继承的父类中的方法。在这个例子中，因为只需显示一个静态网页，只要继承类 `TemplateView`，再提供一个名字是 `template_name` 的 attribute，它指定该类使用的具体的模板名字，就是要加载的网页名，一切就完成了。当用户请求的网址是“根目录/warning”时，Django 就会调用基类的 `as_view()` 方法将这个网页显示出来。

与第二个 URLpattern 对应的视图函数相比较，第一个视图函数的实现要复杂一些。

```
class ProjectOverview(usage.UsageView):  
    table_class = usage.ProjectUsageTable  
    usage_class = usage.ProjectUsage  
    template_name = 'project/overview/usage.html'  
    csv_response_class = ProjectUsageCsvRenderer  
  
    def get_data(self):  
        super(ProjectOverview, self).get_data()  
        return self.usage.get_instances()
```

视图类 `ProjectOverview` 声明了视图函数使用的模板名，属性 `template_name` 指定为“project/overview/usage.html”。看看这个稍微复杂的模板的内容。


```

{% extends 'base.html' %}
{% load i18n %}
{% block title %}{% trans "Instance Overview" %}{% endblock %}

{% block page_header %}
    {% include "horizon/common/_page_header.html" with title=_("Overview") %}
{% endblock page_header %}

{% block main %}
    {% include "horizon/common/_limit_summary.html" %}

    {% if simple_tenant_usage_enabled %}
    {% include "horizon/common/_usage_summary.html" %}
    {{ table.render }}
    {% endif %}
{% endblock %}

```

我们知道 Django 的模板用于产生 HTML 文件和其他各种基于文本格式的文档。模板中通常包含变量和标签。变量用两个大括号括起来——“{{ }}”，标签用一个大括号和一个百分号括起来——“{% %}”。

Django 模板系统支持模板继承与重载。上面的模板里，“{% extend %}”标签表明继承一个基础模板“base.html”，整个网站大部分网页都继承自这个基础模板。“block title”“block page_header”“block main”标签分别表示该模板要重载基础模板网页中有这 3 个标签的网页部分。“include”标签表示在网页的当前位置上包含其他的模板渲染。在“block main”标签中通常显示的是网页的主体内容，在这里我们看到了{{ table.render }}，这表示网页的这部分是要调用一个表格实例的 render 方法来渲染的。在该模板中，simple_tenant_usage_enabled 是一个与该视图相关的上下文变量（Context），标签{% if %}表示要依据这个变量的值选择该网页是否使用这个表格实例去渲染。

现在我们知道了当前网页视图的主要渲染内容其实是一个表格，我们看看 horizon 模块中实现表格视图的流程。在本例中，入口函数是 ProjectOverview.as_view()，as_view()方法在 Django 提供的视图基类（/usr/local/lib/python2.7/dist-packages/django/views/generic/base.py）中实现。

```

class View(object):
    @classmethod
    def as_view(cls, **kwargs):
        # 请求与回应的主要入口点
        for key in kwargs:
            if key in cls.http_method_names:
                raise TypeError("You tried to pass in the %s method name as a "
                                "keyword argument to %s(). Don't do that."
                                % (key, cls.__name__))

```

```

        if not hasattr(cls, key):
            raise TypeError("%s() received an invalid keyword %r. as_view "
                            "only accepts arguments that are already "
                            "attributes of the class." % (cls.__name__,
key))

    def view(request, *args, **kwargs):
        self = cls(**initkwargs)
        if hasattr(self, 'get') and not hasattr(self, 'head'):
            self.head = self.get
        self.request = request
        self.args = args
        self.kwargs = kwargs
        return self.dispatch(request, *args, **kwargs)

    # take name and docstring from class
    update_wrapper(view, cls, updated=())

    # and possible attributes set by decorators
    # like csrf_exempt from dispatch
    update_wrapper(view, cls.dispatch, assigned=())
    return view

    def dispatch(self, request, *args, **kwargs):
        if request.method.lower() in self.http_method_names:
            handler = getattr(self, request.method.lower(),
self.http_method_not_allowed)
        else:
            handler = self.http_method_not_allowed
        return handler(request, *args, **kwargs)

```

`as_view()`方法将 `http` 请求和传给视图函数的参数一并传给 `view()`方法,最后调用 `dispatch()`方法,把 `http` 请求中相应的请求绑定到该视图类 `ProjectOverview` 的方法。在本例中, `dispatch()`方法调用的 `handler` 函数实际上是调用了类 `ProjectOverview` 实例的 `get()`方法。

再看前面显示过的视图代码中,类 `ProjectOverview` 继承了类 `usage.UsageView`。`ProjectOverview` 声明了 3 个属性,一个是 `table_class`,它是一个表格类,指定为 `usage.ProjectUsageTable`。另外两个属性是 `usage_class` 和 `csv_response_class`。此外还定义了一个方法 `get_data()`。不过这个类并没有实现 `as_view()`中要调用的 `get()`方法。

类 `UsageView` 的实现是在 `openstack_dashbaord` 下的 `usage` 模块中的。该模块的主要文件有 `views.py`、`base.py` 和 `table.py`。`views.py` 中是 `UsageView` 视图类的实现,下面是它的声明和初始化函数。

```

from horizon import exceptions

```

```

from horizon import tables
from openstack_dashboard import api
from openstack_dashboard.usage import base

class UsageView(tables.DataTableView):
    usage_class = None
    show_deleted = True
    csv_template_name = None
    page_title = _("Overview")

    def __init__(self, *args, **kwargs):
        super(UsageView, self).__init__(*args, **kwargs)
        .....

```

UsageView 类继承自 horizon.tables.views.DataTableView 类，UsageView 类的初始化函数调用的是它的父类 DataTableView 的初始化函数。DataTableView 类又继承自 MultiTableView，MultiTableView 又继承自 MultiTableMixin 和 Django 中的 generic.TemplateView。最后这个初始化函数的任务是给类的各种关键属性赋初值。

一路跟踪下来，收到 HTTP 请求之后，只有 ProjectOverview 的一个父类 MultiTableView 实现了处理 HTTP 请求的“GET”和“POST”方法，就是下面这个 get() 方法。

```

class MultiTableView(MultiTableMixin, generic.TemplateView):
    def get(self, request, *args, **kwargs):
        handled = self.construct_tables()
        if handled:
            return handled
        context = self.get_context_data(**kwargs)
        return self.render_to_response(context)

```

对本例中 get() 方法而言，它是一个表格视图类的方法，拿到请求后的第一件事，是得到该视图要渲染的表格。所以它调用了 construct_tables() 方法。

```

def construct_tables(self):
    tables = self.get_tables().values()
    for table in tables:
        preempted = table.maybe_preempt()
        if preempted:
            return preempted
    for table in tables:
        handled = self.handle_table(table)
        if handled:
            return handled
    return None

```

方法 `construct_tables()` 调用 `get_tables()` 去拿它的子类声明的表格对象。通过继承关系可以看出，调用的是类 `DataTableView` 的 `get_tables()` 的函数。

```
def get_tables(self):
    if not self._tables:
        self._tables = {}
        if has_permissions(self.request.user,
                           self.table_class._meta):
            self._tables[self.table_class._meta.name] =
self.get_table()
    return self._tables
```

该函数又调用 `get_table()` 函数获取视图所对应的表格类名与类实例的映射，表格的类名从哪里来呢？前面看到过，当前使用的视图类是 `ProjectOverview`，在这个类中声明了 `table_class` 是 `usage` 模块中的类 `ProjectUsageTable`，该类的声明在 `openstack_dashboard/usage/tables.py` 文件中。

```
class ProjectUsageTable(BaseUsageTable):
    instance = tables.Column('name',
                             verbose_name=_("Instance Name"),
                             link=get_instance_link)
    .....
    class Meta:
        name = "project_usage"
        verbose_name = _("Usage")
        columns = ("instance", "vcpus", "disk", "memory", "uptime")
        table_actions = (CSVSummary,)
        multi_select = False
```

这段代码声明 `ProjectUsageTable` 类继承自 `BaseUsageTable` 类。它声明了一个成员类 `Meta`。`Meta` 类描述了该表格类的基本信息，包括表格类的名称，该表格包含的“column”的名称和表格的响应动作等属性。对 `Horizon` 的实现来说，很多类似的视图渲染的实体类都有这样的 `Meta` 成员类，它一般放的都是该实体类抽象出来的特殊属性。从这段代码中可以找出上述 `get()` 方法想要获取的表格类对应的 `Meta` 类名称。

下面是 `horizon.tables.views.DataTableView` 类里 `get_table()` 函数的实现。

```
def get_table(self):
    if not self.table_class:
        raise AttributeError('You must specify a DataTable class for the '
                              '"table_class" attribute on %s.'
                              % self.__class__.__name__)
    if not hasattr(self, "table"):
        self.table = self.table_class(self.request, **self.kwargs)
    return self.table
```


这段代码先做属性检查,如果该视图类属性里的 table 实例还没有生成,则先实例化该 table 类。openstack_dashboard.usage.tables.ProjectUsageTable 类继承自 BaseUsageTable 类, BaseUsageTable 类又继承自 horizon.tables.base.DataTable, 具体的表格类实现包括 row, cell 的实现以及相应的 action 动作, 具体可参见 <http://wiki.openstack.org/developer/horizon/ref/tables.html>, 这里我们只分析源码的走向, 下面看 DataTable 的实例化代码。

```
def __init__(self, request, data=None, needs_form_wrapper=None,
**kwargs):
    self.request = request
    self.data = data
    self.kwargs = kwargs
    self._needs_form_wrapper = needs_form_wrapper
    self._no_data_message = self._meta.no_data_message
    self.breadcrumb = None
    self.current_item_id = None
    self.permissions = self._meta.permissions

    columns = []
    for key, _column in self._columns.items():
        column = copy.copy(_column)
        column.table = self
        columns.append((key, column))
    self.columns = SortedDict(columns)
    self._populate_data_cache()

    for action in self.base_actions.values():
        action.associate_with_table(self)

    self.needs_summary_row = any([col.summation
                                  for col in self.columns.values()])
```

这段代码初始化表格的一些相关属性和数据, 比如传入的 HTTP 请求, 访问表格所需的权限, 该表格里是否有表单等。该代码还将前面两个父类 ProjectUsageTable 和 BaseUsageTable 里分别声明的多个 columns 整合在自己的 columns 变量中, 和表格相关的动作 action 也在这里与表格相关联。

在表格的初始化动作完成后, 该 horizon.tables.views.DataTableView 视图类终于关联到它要渲染的表格实体类了。跟着函数调用的回滚, 代码流又回到 horizon.tables.views.MultiTableView 类的 construct_tables 函数。表格类已经具备, 现在要考虑的是怎么获取该表格的数据。在前面已经列出的 construct_tables 方法中, horizon.tables.base.DataTable 类的 maybe_preempt() 方法检测是否表格的数据已经加载过了, 如果没有, 父类 MultiTableMixin 视图类的 handle_table() 方法加载表格数据并且依次调用表格 actions 的处理函数。


```
def handle_table(self, table):
    name = table.name
    data = self._get_data_dict()
    self._tables[name].data = data[table._meta.name]
    self._tables[name]._meta.has_more_data = self.has_more_data(table)
    self._tables[name]._meta.has_prev_data = self.has_prev_data(table)
    handled = self._tables[name].maybe_handle()
    return handled
```

`handle_table()`方法最后调用视图类的 `_get_data_dict()`方法以获取该表格的数据。这个方法在 `DataTableView` 视图类里实现如下所示。

```
def _get_data_dict(self):
    if not self._data:
        self.update_server_filter_action()
        self._data = {self.table_class._meta.name: self.get_data()}
    return self._data
```

如上述代码所示, `self._data` 是一个指定表格元类名为 `key` 值的数据字典。这会调用该视图类的 `get_data()`函数。当前的表格元类的属性名字是 `project_usage`。`DataTableView` 视图类没有实现 `get_data()`方法, 它的子类 `ProjectUsageView` 视图类实现了该方法。

```
class ProjectOverview(usage.UsageView):
    def get_data(self):
        super(ProjectOverview, self).get_data()
        return self.usage.get_instances()
```

由 `super()`函数又回滚追踪到它的父类 `UsageView` 视图类。

```
class UsageView(tables.DataTableView):
    def get_data(self):
        try:
            project_id = self.kwargs.get('project_id',
                                          self.request.user.tenant_id)
            self.usage = self.usage_class(self.request, project_id)
            self.usage.summarize(*self.usage.get_date_range())
            self.usage.get_limits()
            self.kwargs['usage'] = self.usage
            return self.usage.usage_list
        .....
```

`ProjectOverview` 视图类要渲染的表格的数据来源是另一个专门处理 `usage` 数据的类来处理的。回忆一下在前面显示的代码中, `ProjectUsageView` 声明了两个类, 即 `table class` 和 `usage_class`。`usage_class` 类名是 `ProjectUsage`, 它又以 `BaseUsage` 为父类。上述代码中, `usage_class` 的初始化, 取得 `usage` 信息的种种实现都在 `openstack_dashboard/usage/base.py` 中。这些信息一般都关系到 OpenStack 的其他服务, 比如 `self.usage.get_limits()`用来获取 OpenStack

网站提供的各种资源的限制。跟踪它的代码看看它的实现。

```
class BaseUsage(object):
    def get_limits(self):
        try:
            self.limits = api.nova.tenant_absolute_limits(self.request)
        except Exception:
            exceptions.handle(self.request,
                              _("Unable to retrieve limit information."))
        self.get_neutron_limits()
        self.get_cinder_limits()
```

这个方法的实现在 `BaseUsage` 类里，看起来十分直观，它调用 OpenStack 其他服务提供的接口，以获取需要渲染的数据。`api.nova.tenant_absolute_limits()`通过 Nova Client 调用 Nova API 以取得数据，类似地，`self.get_neutron_limits()`函数和 `self.get_cinder_limits()`方法则分别向 Neutron 和 Cinder 服务提出请求。这个 API 模块的实现在 `openstack_dashboards/api` 目录下。

现在 `ProjectOverview` 类要渲染的表格实例已经获得了要显示的数据。到此，前文提到的视图类的构造表格的函数 `construct_tables()`已经走完。代码回滚，调用这个函数的地方是视图类 `MultiTableView` 的 `get()`函数，我们还记得，这是本次 HTTP 请求入口处。表格数据已经准备好了，接下来的代码要处理网页的渲染。

```
class MultiTableView(MultiTableMixin, generic.TemplateView):
    def get(self, request, *args, **kwargs):
        handled = self.construct_tables()
        if handled:
            return handled
        context = self.get_context_data(**kwargs)
        return self.render_to_response(context)
```

在 Django 框架的模板系统中，模板中的变量是通过参数 `Context` 传递的。在用模板渲染网页之前，模板中的变量必须赋值。所以接下来的代码要把前面代码中准备的数据赋值给 `Context` 中指定的上下文变量。

接下来的代码要调用 `get_context_data()`函数。这是为模板准备 `Context` 上下文的方法。`ProjectOverview` 视图类没有实现该函数，调用的是 `UsageView` 视图类的 `get_context_data()`方法。该方法把能在该视图类中获取的数据按 `key` 值赋给 `Context` 中的变量，再用 `super()`函数继续回滚调用父类的 `get_context_data()`函数，把每个父类中获得的 `Context` 中的变量赋值，一直回滚到 Django 的基类，最后得到一个 Django 能够处理的完整的 `Context` 上下文。

```
class UsageView(tables.DataTableView):
    def get_context_data(self, **kwargs):
        context = super(UsageView, self).get_context_data(**kwargs)
        context['table'].kwargs['usage'] = self.usage
        context['form'] = self.usage.form
```

```

context['usage'] = self.usage
try:
    context['simple_tenant_usage_enabled'] = \
        api.nova.extension_supported('SimpleTenantUsage',
self.request)
except Exception:
    context['simple_tenant_usage_enabled'] = True
return context

```

这个例子中的 Context 上下文字典打印出来如图 11-4 所示。

```

Context:
{
  'project_usage_table': <ProjectUsageTable: project_usage>,
  'form': <horizon.forms.base.DateForm object at 0x7faa88373e10>,
  'simple_tenant_usage_enabled': True,
  'usage': <openstack_dashboard.usage.base.ProjectUsage object at 0x7faa88373f50>,
  'table': <ProjectUsageTable: project_usage>,
  'u'view': <openstack_dashboard.boards.project.overview.views.ProjectOverview object at 0x7faa88390c90>
}

```

图 11-4 上下文字典

回忆一下当前 Horizon 要加载的模板文件，那里面主要有两个变量需要 Context 上下文提供参数：一个是 `simple_tenant_usage_enabled`，另一个是 `table.render`。前一个变量值在 Context 中已经赋值为 `True`，后一个变量表示的是要调用 Context 变量中的 `table` 实例的 `render` 方法。在本例中，该 `table` 实例在 Context 中也已赋值，Django 调用模板渲染时就会调用该 `table` 实例的 `render` 方法。

获取了模板的 Context 上下文之后，代码要流向 Django 提供的模板渲染机制，本例中一次性地载入模板，渲染，最后返回 `HttpResponse`。看看上文 `UsageView` 视图类显示的 `render_to_response()` 方法。

```

class UsageView(tables.DataTableView):
    def render_to_response(self, context, **response_kwargs):
        if self.request.GET.get('format', 'html') == 'csv':
            render_class = self.csv_response_class
            response_kwargs.setdefault("filename", "usage.csv")
        else:
            render_class = self.response_class
        context = self.render_context_with_title(context)
        resp = render_class(request=self.request,
                           template=self.get_template_names(),
                           context=context,
                           content_type=self.get_content_type(),
                           **response_kwargs)

        return resp

```

这段代码根据收到的 `HttpRequest` 的内容选择渲染类。如果是请求 `csv` 文件格式，渲染类就选作 `csv_response_class`，该类在 `ProjectOverview` 类中声明过。反之，就选 Django 提供的渲

染类——`django.template.response.TemplateResponse` 类。它的主要参数是该网页请求，模板名字和网页内容的类型等。接下来，Django 会加载这个模板，调用该模板中的 `table.render` 方法。

有一点稍微说明一下，就是模板的加载。前文我们提到过，`horizon` 提供自己的模板加载器，主要用来加载 Dashboard 下各个 `panel` 模块的模板，该加载器列在文件 `openstack_dashboard/settings.py` 的 `TemplateLoader` 变量下。

```
TEMPLATE_LOADERS = (
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader',
    'horizon.loaders.TemplateLoader'
)
```

当各个 Dashboard 注册它所属的 Panel 时，把模板的“`templates`”目录拼接在 `panel` 所在的路径上，并将这个路径记录到全局的变量 `panel_template_dirs` 中。现在回想本例中用到的模板文件名在 `ProjectOverview` 类中声明为“`project/overview/usage.html`”。Django 怎么找到这个模板呢？看看文件 `horizon/loaders.py` 中 `TemplateLoader` 的实现，它将模板的名字以“/”为界分成 Dashboard 名、Panel 名以及文件名 3 个部分。先以“`dashboard`”/“`Panel`”为 key 值，在变量 `panel_template_dirs` 中查找，找到相应的模板路径名后，再拼接上文件名。其实就是把模板生成的路径与模板加载的方法做了一个简单映射。

在本例中，`table.render()` 方法是调用的父类 `horizon.tables.base.DataTable` 的 `render()` 方法。

```
def render(self):
    table_template = template.loader.get_template(self._meta.template)
    extra_context = {self._meta.context_var_name: self,
                     'hidden_title': self._meta.hidden_title}
    context = template.RequestContext(self.request, extra_context)
    return table_template.render(context)
```

实际上这个方法又加载了它自己的模板，这是 `horizon` 模块为 `DataTable`（数据类表格）提供的通用模板。`self._meta.template` 赋值为 `horizon/common/_data_table.html`，模板放在 `horizon/templates/horizon/common` 的目录下。

```
{% load i18n %}
{% with table.needs_form_wrapper as needs_form_wrapper %}
<div class="table_wrapper">
    {% if needs_form_wrapper %}<form action="{{ table.get_full_url }}"
method="POST">{% csrf_token %}{% endif %}
    {% with columns=table.get_columns rows=table.get_rows %}
    {% block table %}
        <table id="{{ table.slugify_name }}" class="{% block
table_css_classes %}tab">
            <thead>
                {% block table_caption %}
```

```

<tr class='table_caption'>
  <th class='table_header' colspan='{{ columns|length }}'>
    <h3 class='table_title'>{{ table }}</h3>
    {{ table.render_table_actions }}
  </th>
</tr>
{% endblock table_caption %}
.....

```

这个模板规范了 Horizon 生成的网站渲染的数据表格的具体样式，看起来有点复杂。模板中到处都在调用通过 Context 上下文传进的 table 实例的各种方法，借此来实现 row、column、cell、action、caption、footer 等表格元素的渲染。DataTable 类中的主要代码就是这些方法的实现。除了这个模板，这个目录下还提供了除了表格外的其他一些渲染的实体的基本模板，比如表单、工作流、标签等。

代码分析走到这里，我们就看见了网页上“overview”Panel 显示的网页。

从 2013 年开始，容器技术随着 Docker 的出现迅速成为广大互联网厂商的平台服务的首选。虽然 Docker 技术并不是十分完善与稳定，但容器技术在未来的发展趋势已经势不可挡。

随着容器技术的发展，容器与云基础架构的结合受到越来越多的关注，OpenStack 也在不断发展来对其进行支持。为了更好地说明容器现下如此广受关注的原因，OpenStack 基金会于 2015 年发布了一篇名为《探索基于：容器与 OpenStack》的白皮书 (<http://www.openstack.org/containers>)，详细地介绍了在 OpenStack 中容器的价值，并给出了一些容器的使用案例。

12.1 容器技术

容器技术又被称为操作系统级别的虚拟化，与虚拟化技术相比，两者都提供了资源的隔离访问，但从原理上有根本的不同。容器系统不需要运行客户机操作系统，它共享主机操作系统内核，利用各种操作系统级别的隔离技术实现容器之间的资源访问隔离。

容器技术并不是新兴的技术，在 1972 年 UNIX 就提出通过 chroot 修改操作系统根目录，并隔离其他进程访问的方式来提供磁盘的隔离，这是容器技术的雏形。2013 年，随着 Docker 容器技术的出现，其提供的容器镜像制作与发布方式有改变软件发布方式的趋势，使得运维更加容易和方便，软件可维护性大大增强，容器技术再次引起了人们的重视。除了 Docker 容器技术，其他容器技术包括 Ubuntu 的 LXC/LXD，CloudFoundry 的 Warden，CoreOS 公司出品的 Rocket，还有其他使用虚拟机方式运行容器的 Intel Clear Container 项目，国内创业公司 Hyper 的 Hyper container 等。随着近年来的逐步发展进化，容器技术开始在生产环境中大批量部署并被用于运行各种服务。

12.1.1 容器的原理

容器技术的核心是资源隔离，同时又共享内核空间某些系统运行时的数据，但最终访问系统硬件资源还是通过主机的内核完成的。在现代 Linux 内核中容器使用了以下技术。

- 用户/组特权隔离：通过对应用进程设置不同的用户 ID 或者组 ID 来实现资源访问的限制，即 A 用户创建的资源（文件、目录等）对 B 用户不可见。
- 文件系统隔离：使用 chroot 的方式对文件系统进行隔离。
- 控制组资源隔离：Cgroup 提供了对操作系统内存、CPU、I/O 资源使用的限制，可对

某一进程能获得的最大资源进行限定。

目前 Linux 内核中已经实现了多种可用于容器隔离技术的命名空间。

- Mount 命名空间：提供 Linux 目录挂载点的隔离访问。
- UTS 命名空间：为容器提供了主机名和域名的隔离。
- IPC 命名空间：用于隔离不同进程间的通信。
- PID 命名空间：对进程空间进行隔离，使得不同命名空间的两个进程有多个相同的进程号。
- Network 命名空间：隔离不同容器的网络，使得不同容器可以具有相同的网络资源，如两个同一主机上的容器在自己的 Network 命名空间中可以看到相同端口号。
- User 命名空间：可以在新的用户命名空间中创建拥有特权的超级用户。比如，特权用户的 user ID 是 0，当创建新的容器实例时在新的用户空间中，用户的 user ID 也可以是 0，在容器内部，此用户是特权用户；而在容器外部，该用户仍然是普通用户。

有关各种命名空间可以参考 Linux 官方文档 <http://lwn.net/Articles/531114/>。

12.1.2 常见的容器集群管理工具

我们可以在单独主机上用容器来运行应用程序，然而对于企业用户或者数据中心而言，想要大规模部署和使用容器必须使用得心应手的集群管理工具以提供各种企业级的服务支撑。

随着 Docker 将容器技术的易用性提升，用于编排 Docker 容器的管理工具应运而生。这里仅介绍几种目前主流的集群编排工具：Docker Swarm、Kubernetes 和 Mesos。

1. Docker Swarm

Docker Swarm 是 Docker 公司自己推出的 Docker 容器集群管理工具。通过 Docker Swarm，管理员可以将多台 Docker 主机组成一个 Docker 容器集群，就像使用一台 Docker 主机一样使用整个 Docker 容器集群。

图 12-1 所示为 Docker Swarm 的系统架构。Docker Swarm 包含了一个（或多个 HA 模式的）Swarm Master 节点，以及多个 Swarm Node 节点。Swarm Master 包含了集群的管理功能，如新建容器的调度策略、集群健康监控等。Swarm Master 通过与 Swarm Node 的特定 TCP 或者 HTTP 服务端口号（2378）进行通信来控制 Swarm Node 节点。

Swarm 集群的优点是集群管理员可以使用 Docker 客户端直接访问 Docker Swarm 集群的服务端口来控制整个集群，其中，Docker Swarm 的 API 几乎 100%兼容原生的 Docker API。

在 Docker 1.12 版本之前，如果想使用 Docker Swarm 集群，各节点除了需要安装 Docker Engine 服务，我们还需要单独部署 Swarm Master 和 Swarm Node 服务（Swarm Node）。从 Docker 1.12 版本开始，Docker Swarm 的功能已经集成到了 Docker Engine 服务中，只需要开启 Swarm 模式，不需要安装额外的 Docker Swarm 服务就可以直接组建 Docker 集群了。具体细节可以参考 <https://docs.docker.com/engine/swarm/>。

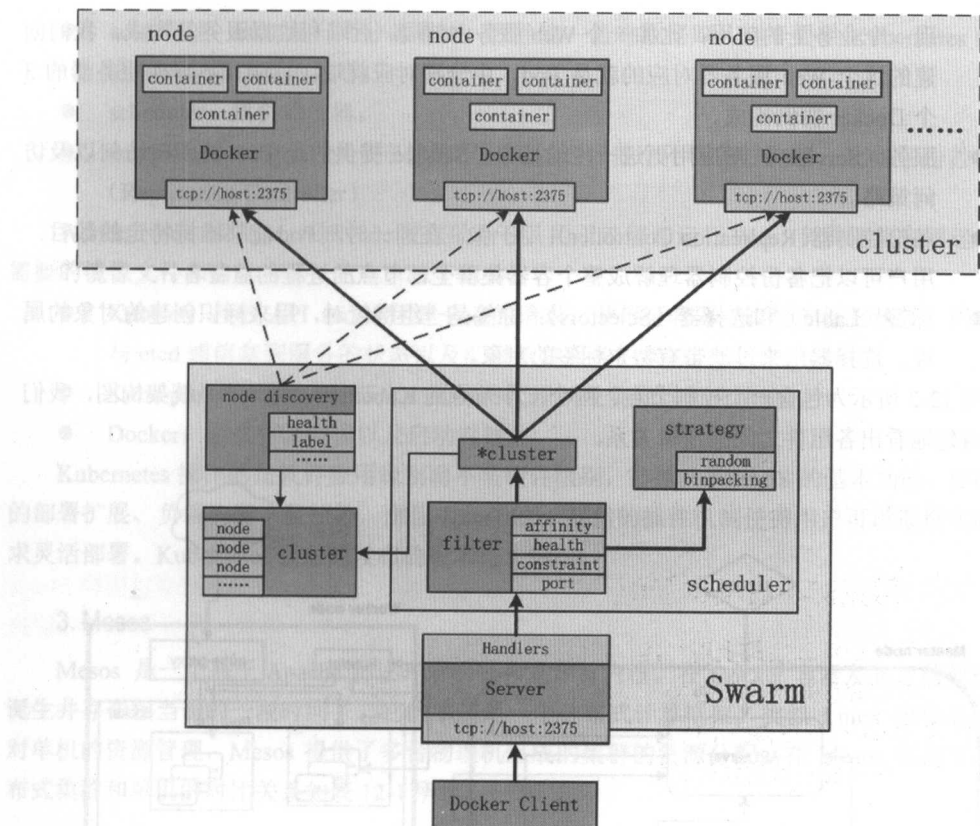


图 12-1 Docker Swarm 系统架构

2. Kubernetes

Kubernetes 是谷歌公司的开源大规模集群管理工具。Kubernetes 一词源于希腊语，它的含义是舵手，相对于 Docker 的集装箱含义，Kubernetes（舵手）象征着对 Docker 的完美控制。

Kubernetes 自 2014 年 6 月由谷歌宣布开源到现在，已经有包括微软、Red Hat、IBM 等在内的诸多 IT 巨头加入社区贡献行列。Google 于 2015 年 7 月宣布加入 OpenStack 基金会，希望把容器技术带入 OpenStack，用于提高公有云和私有云的互操作性。

为了更方便地适用大规模部署和管理，Kubernetes 为 Docker 容器提出了更高层次的抽象，包含节点（Node）、Pod、服务（Service）、备份控制器（Replication Controller）、标签（Label）和选择器（Selectors）等。

- 节点：或者叫 Slave（之前称为 Minion），运行 Pod 实例的载体，可以为物理机或者虚拟机。
- Pod：Kubernetes 集群中控制和管理的最小单元。它是对运行的应用的抽象，通常一个 Pod 由一个或者多个相关联的 Docker 实例组成，这些 Docker 实例共同协作来实

现一个业务上的应用。比如一个 Web 服务由前端、后端和数据服务器构成，我们创建的这个 Web 服务器对应的就是 Pod，由分别对应前端、后端和数据库服务器的 3 个 Docker 实例构成。

- 服务 (Service)：对应用更进一步的抽象。Service 提供的是 Pod 的入口访问以及访问策略。
- 备份控制器 (Replication Controller)：用于保证在同一时刻 Pod 能够维持特定的数目。用户可以把备份控制器理解成整个容器集群全部节点的进程的监督者。
- 标签 (Label) 和选择器 (Selectors)：标签是一组键值对，用来标识创建的对象属性。选择器用来过滤带有特定标签的对象。

图 12-2 所示为包含一个控制节点，两个工作节点的 Kubernetes 集群的系统架构图，我们可以清楚地看出各组件之间的逻辑关系。

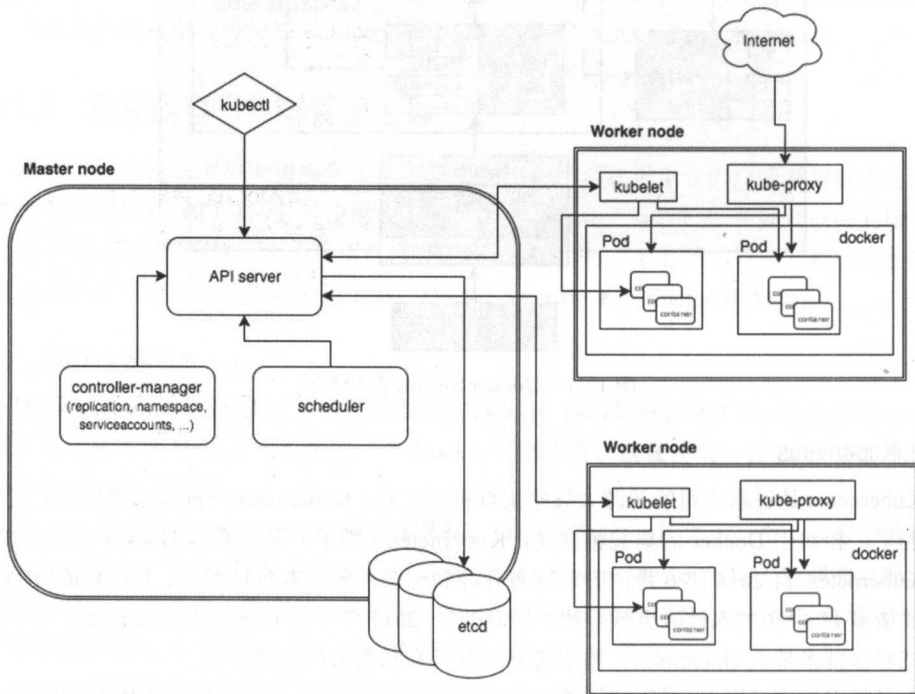


图 12-2 Kubernetes 系统架构

Master Node 提供了管理整个 Kubernetes 集群的功能，也是所有管理员控制集群的入口，并提供 Worker Node 的编排。Master Node 包含了以下组件。

- API server：提供 REST API 服务，用于处理 REST API 请求、参数验证等，相应的处理请求将会被持久化到 Kubernetes 的数据库中。

- **etcd 存储**: etcd 是一个分部署的一致性键—值数据库, 用于提供 Kubernetes 组件之间的配置信息的共享。
- **scheduler**: 集群调度器。
- **controller-manager**: 一个守护进程, 可以运行多个不同的控制器, 比如备份控制器 (Replication Controller)

Worker Node 接收 Master Node 发送的指令, 用于部署和运行 Pod, 负责下载容器运行所需要的镜像文件和启动容器, 它包含以下组件。

- **Kubelet**: 接收 API server 创建 Pod 的请求, 并保证容器一直处于运行状态。同时也与 etcd 通信拿到服务的状态以及汇报新建服务的状态。
- **Kube-proxy**: 提供工作节点上的网络代理以及负载均衡的功能。
- **Docker**: 负责下载镜像以及启动容器。

Kubernetes 操作的是软件应用级别而不是硬件级别, 它提供了 PASS 的基本功能, 如应用的部署扩展、负载均衡、监控等, 而且 Kubernetes 提供的插件机制使得用户可以根据实际需求灵活部署。Kubernetes 目前的发布的版本是 1.3.0。

3. Mesos

Mesos 是一个基于 Apache 协议开发的分布式计算内核。在 Docker 容器发布之前就已经诞生并存在相当长的一段时间了。它的本质是一个分布式计算框架, 类似 Linux 内核提供了对单机的资源管理, Mesos 提供了多台物理机组成的集群的资源分配。在 Mesos 框架下, 分布式集群和单机的对比关系如表 12-1 所示。

表 12-1 Linux 操作系统与分布式操作系统的对比

	Linux 操作系统	分布式操作系统
资源管理	Linux Kernel	Mesos
应用执行	Linux Kernel	Docker
进程管理	Init.d	Marathon, Chronos
进程/应用间通信	管道、套接字、IPC	消息队列
文件系统	Linux 文件系统	各种分布式文件系统

Mesos 生态更关注的是如何对多主机的资源进行调配, 其中 Mesos 相当于分布式系统的核心, Mesos 的架构如图 12-3 所示, Mesos Master 负责资源的分配, Mesos Slave 负责应用程序的执行与资源使用情况的上报, ZooKeeper 提供了高可用模式下的消息传递。在整个框架之上, 通过 Marathon、Chronos 等软件来做应用程序生命周期的管理。

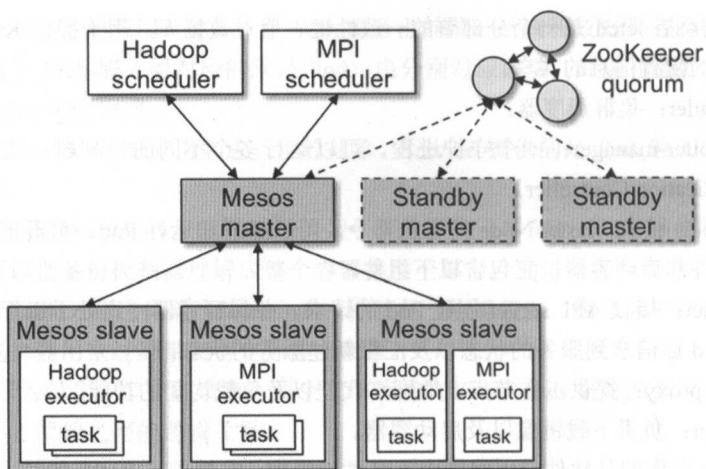


图 12-3 Mesos 系统架构

基于 Mesos, Mesosphere 公司推出了数据中心操作系统 DC/OS, 它是一整套以围绕 Mesos 为中心的工具体集, 用来支持整个数据中心的资源的池化以及管理, 提供一整套数据中心解决方案。

如上所述的 Docker Swarm、Kubernetes 以及 Mesos 3 种工具 (集) 都是用来驱动 Docker 容器的, 分别从不同层面有各自的用途以及特点。

- Docker Swarm 提供了对 Docker API 的完全兼容, 将单一主机的容器运行扩展到多台主机组成的集群。
- Kubernetes 提出了新的应用程序管理的抽象, 简化了应用的运维部署。
- Mesos 则是新型的分布式数据中心系统。

12.2 容器与 OpenStack

随着容器技术的迅猛发展, 有关“容器技术是不是会替代 OpenStack”的问题不时被人提起。容器技术更是在 2015 年的温哥华 Summit 上成为关注的热点之一。OpenStack 基金会首席运营官 Mark Collier 在自己的主题演讲中花了大量时间对相关话题进行了论述。他认为, 用户应该将 OpenStack 视为一个整合引擎, 与前些年将许多不同虚拟层进行整合, 从而帮助开发者管理虚拟机的项目一样, OpenStack 社区也将接受容器和谷歌 Kubernetes 等容器管理平台, 并将它们整合至自己的平台中。

而 OpenStack 社区在 2014 年决定将容器作为需要支持的重要技术, 并从而衍生出了几个项目来支持容器和容器的第三方生态系统。

12.2.1 nova-docker/heat-docker

nova-docker 插件是 OpenStack 和 Docker 的第一次集成，主要是把 Docker 作为一种新的 Hypervisor，把所有的 Container 当成 VM 来处理，通过 Docker REST API 来操作 Container。Nova 与 Docker 的集成架构如图 12-4 所示。

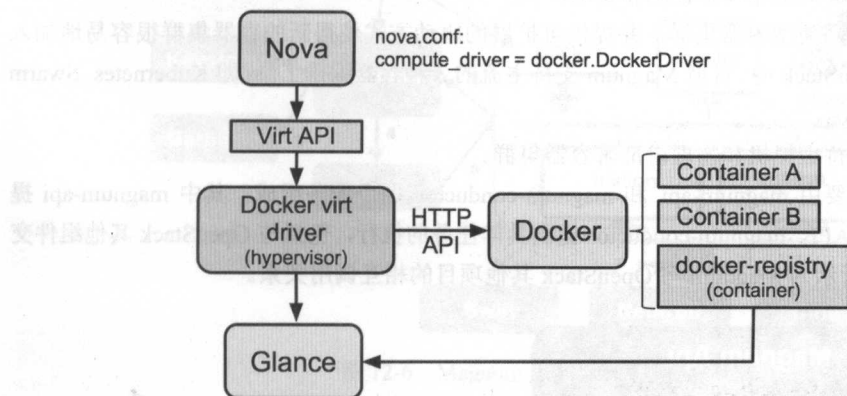


图 12-4 Nova 与 Docker 集成

nova-docker 插件被实现为 Nova 的一个 Virt Driver 加入到了 Nova 中，通过继承 Nova Virt Driver 基类，实现其中对应的方法，将 Nova API 服务转发过来的请求通过 docker-py 以 socket 请求发送 Docker daemon。但因为 Docker 的大部分 API 与 Nova 现有的 API 不兼容，所以最终被废弃了。

因为 Nova Docker Driver 不能使用 Docker 的一些高级功能，所以社区就想了另一个方法，和 Heat 去集成。通过 Heat 的 heat-docker 插件，我们可以把 OpenStack 中的其他资源，比如网络、存储、计算资源进行统一编排，然而这样做只能使用到 Docker 的部分功能，而且基本是静态的操作，所以，这条路最终也没有成功。

12.2.2 Magnum

在 OpenStack 和 Docker 集成的过程中，并不能从 OpenStack 原有的项目中找到一个很好的集成点，虽然在 Nova 以及 Heat 中都做了相当的尝试，但缺点很明显，所以在 2014 年年底 Docker 被广泛提及的时候，社区就开始了一个新的专门针对 Docker 和 OpenStack 集成的项目 Magnum。

最初 Magnum 项目成立的目的是提供 OpenStack 中的 Cass（容器即服务）。Magnum 是第一个与容器发生关系的 OpenStack 项目，在发起后不久就马上成为了 OpenStack 的正式项目。Magnum 充分利用了 OpenStack 中的现有框架以及服务，并吸取了其他成功项目的长处，来实现 OpenStack 上的容器集群的管理。

1. Magnum 体系结构

Magnum 项目利用了 OpenStack 中计算、存储、网络、编排以及认证服务来为 OpenStack 用户提供生产级可用的容器集群管理服务。

Magnum 的核心功能可以概括为以下几个方面：

- 容器集群的生命周期控制和管理。
- 抽象不同类型容器集群，并提供可扩展的驱动方式使得新的容器集群很容易地加入到 OpenStack 中。目前 Magnum 支持主流的 3 种容器编排工具，即 Kubernetes、Swarm 与 Mesos。
- 同时支持虚拟机和物理机部署容器集群。

Magnum 主要由 magnum-api 和 magnum-conductor 两个服务组成，其中 magnum-api 提供对外的 REST API，magnum-conductor 提供具体任务的执行，包括与 OpenStack 其他组件交互等。图 12-5 所示为 Magnum 与 OpenStack 其他项目的相互调用关系。

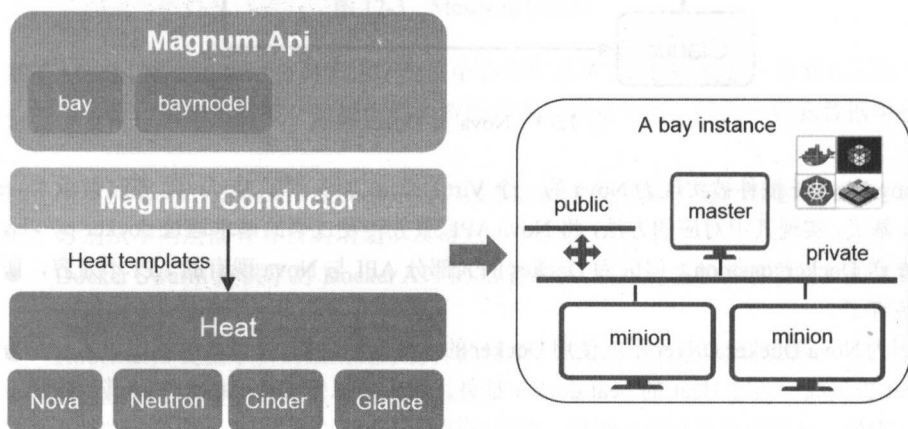


图 12-5 Magnum 系统架构

目前 Magnum 项目中有两个重要的概念：Bay 和 Baymodel。Bay 就是一个容器集群的实例化，为方便理解，最新版本的 Magnum 明确将 Bay 修改为 Cluster（具体原因请参见 <https://blueprints.launchpad.net/magnum/+spec/rename-bay-to-cluster>，为保持兼容性，Bay 和 Cluster 目前在代码中同时存在）。类比 Nova instance 和 flavor 的含义，Bay 是 Baymodel 的一个具体实例，Baymodel 是对 Bay 的抽象。

图 12-5 中，Magnum API 服务通过 RPC 调用将创建容器集群所需要的信息（即实例化一个 Baymodel 所需要的信息，如 COE 类型、VM 的 flavor、master、node 数量、网络驱动等）传递给 Magnum Conductor，生成具体的 Heat 模板。Heat 根据模板定义通过 Nova、Neutron、Cinder 和 Glance 向 OpenStack 集群申请创建集群所需要的计算、网络、存储等资源，并完成容器集群的部署以及初始化功能，用户最终得到右图所示的一个集群实例。

Magnum 与 Heat、Keystone、Barbican 服务都有交互。图 12-6 所示为 Magnum 服务请求处理流程。

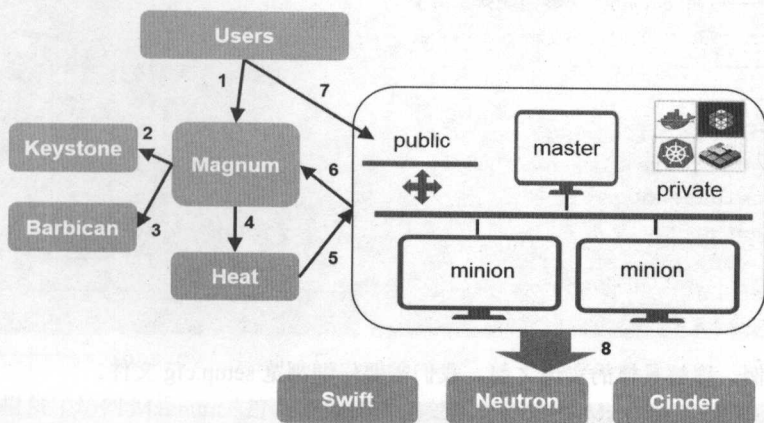


图 12-6 Magnum 用户请求调用流程

该请求过程可概括如下：

- 1) 用户请求创建一个容器集群实例。
- 2) 生成 Cluster 专用的账号（Keystone trust & trustee）。
- 3) 生成根密匙和根证书，储存在 Barbican。
- 4) 用 Heat 模板向 OpenStack 集群申请资源创建集群需要的各种资源。
- 5) Heat 创建 stack。
- 6) 各个节点请求 Magnum 签署证书。
- 7) 用户请求 Magnum 签署证书，然后用密匙访问 Bay 的 API。
- 8) 根据具体的请求，容器集群内部用 Keystone trust 访问 OpenStack 其他的服务。

2. Magnum 源码目录结构

```

.
├── api-ref
├── contrib
├── devstack - 用于使用 devstack 安装
├── doc - 开发者文档目录
├── etc - 配置文件目录
├── install-guide
├── magnum
│   ├── api - Magnum API 服务实现代码
│   ├── cmd - 各种 Magnum 服务的启动程序
│   └── common - 通用库

```

```

|   |—— conductor - conductor 服务的实现
|   |—— conf
|   |—— db - 数据库抽象层
|   |—— drivers - 容器编排引擎驱动
|   |—— hacking
|   |—— objects
|   |—— service
|   |—— servicegroup
|   |—— tests
|—— releasenotes
|—— setup.cfg
|—— setup.py
|—— specs
|—— tools

```

依照惯例，理解具体的实现之前，我们需要仔细浏览 `setup.cfg` 文件。

```

console_scripts =
    magnum-api = magnum.cmd.api:main
    magnum-conductor = magnum.cmd.conductor:main
    magnum-db-manage = magnum.cmd.db_manage:main
    magnum-driver-manage = magnum.cmd.driver_manage:main

```

命名空间 “`console_scripts`” 里，涵盖了 Magnum 所提供的所有服务以及工具，其中的每一行都指示了相应 Magnum 工作的入口，从中可以看到组成 Magnum 的两个主要服务：`magnum-api` 和 `magnum-conductor`。

```

magnum.drivers =
    k8s_fedora_atomic_v1 =
magnum.drivers.k8s_fedora_atomic_v1.driver:Driver
    k8s_coreos_v1 = magnum.drivers.k8s_coreos_v1.driver:Driver
    swarm_fedora_atomic_v1 =
magnum.drivers.swarm_fedora_atomic_v1.driver:Driver
    mesos_ubuntu_v1 = magnum.drivers.mesos_ubuntu_v1.driver:Driver
    k8s_fedora_ironic_v1 =
magnum.drivers.k8s_fedora_ironic_v1.driver:Driver

```

命名空间 “`magnum.drivers`” 里定义了目前 Magnum 所支持的默认容器集群驱动。

3. Magnum API

Magnum API 服务基于 Pcan 框架构建，我们只需关注自己设计的 Controller 方法，开发变得更容易，迭代更快速。

Magnum API 服务位于 `magnum/api` 下，目前支持 V1 版本，具体代码如下：

```

|—— controllers

```



```

@base.Controller.api_version("1.1", "1.1")
# 设置 API 方法返回对象为 Bay, REST API 接受到的请求 body 为 Bay 类型, 函数
# 成功返回状态为 201
@expose.expose(Bay, body=Bay, status_code=201)
def post(self, bay):
    new_bay = self._post(bay) # 定义一个新的 Bay 对象
    # 调用 rpcapi 对 cluster_create 方法向 Magnum Conductor 请求
    # 创建一个新的 Bay 实例
    res_bay = pecan.request.rpcapi.cluster_create(new_bay,
                                                    bay.bay_create_timeout)

    # 拼接 pecan 响应的 location 信息
    pecan.response.location = link.build_url('bays', res_bay.uuid)
    # 将 Bay 实例对象转化为 REST API 对象返回 HTTP 调用方
    return Bay.convert_with_links(res_bay)

```

Magnum API 服务同样提供了 policy 检查, policy 设置存放在源码目录的 etc/magnum/policy.json 文件中, 同目录下的 api-paste.ini 文件定义了一些 Magnum REST API 服务用到的中间件服务。

4. Magnum Conductor

由 setup.cfg 文件可知, magnum-conductor 服务的入口为 magnum.cmd.conductor, 作为一个服务协调者, Magnum Conductor 接收 Magnum API 服务的 RPC 请求, 它是一个 RPC API 服务器, 注册了 4 个处理 endpoints:

```

endpoints = [
    indirection_api.Handler(), # 间接 API 调用, 处理所有 Magnum object
                                # 对象的方法调用
    cluster_conductor.Handler(), # 处理和 cluster 有关的操作, 如
                                # bay/bay_model 的操作
    conductor_listener.Handler(), # 用于响应心跳检测
    ca_conductor.Handler(), # 处理证书授权等相关的操作
]

```

cluster_conductor.Handler() 为主要处理容器集群的服务代码, 其承担着对集群的管理功能, 以创建集群为例, 具体代码如下:

```

def cluster_create(self, context, cluster, create_timeout):
    osc = clients.OpenStackClients(context)

    try:
        # Create trustee/trust and set them to cluster
        trust_manager.create_trustee_and_trust(osc, cluster)
        # Generate certificate and set the cert reference to cluster

```

```

cert_manager.generate_certificates_to_cluster(cluster,
                                              context=context)
conductor_utils.notify_about_cluster_operation(
    context, taxonomy.ACTION_CREATE, taxonomy.OUTCOME_PENDING)
# Get driver
ct = conductor_utils.retrieve_cluster_template(context, cluster)
cluster_driver = driver.Driver.get_driver(ct.server_type,
                                          ct.cluster_distro,
                                          ct.coe)

# Create cluster
created_stack = cluster_driver.create_stack(context, osc, cluster,
                                             create_timeout)
except Exception as e:
    cluster.status = fields.ClusterStatus.CREATE_FAILED
    cluster.status_reason = six.text_type(e)
    cluster.create()
    conductor_utils.notify_about_cluster_operation(
        context, taxonomy.ACTION_CREATE, taxonomy.OUTCOME_FAILURE)

    if isinstance(e, exc.HTTPBadRequest):
        e = exception.InvalidParameterValue(message=six.text_type(e))

        raise e
    raise

cluster.stack_id = created_stack['stack']['id']
cluster.status = fields.ClusterStatus.CREATE_IN_PROGRESS
cluster.create()

self._poll_and_check(osc, cluster, cluster_driver)

return cluster

```

Magnum Conductor 通过 OpenStack Client 对象来调用 OpenStack 其他服务。首先创建 OpenStack Client (osc)，并创建一个证书，该证书用于提供之后 cluster 访问 OpenStack 服务的认证信息。然后获取所创建集群的模板 ct，根据 ct 得到集群驱动，调用集群驱动的 create_stack 方法向 Heat 服务发送创建 Stack 请求。最后调用 _poll_and_check() 方法去同步 cluster 在 Heat 服务中的状态，并更新 Magnum 的 cluster 状态与之同步。

5. Magnum 集群驱动

为更灵活地支持不同类型的集群，Magnum 提出了集群驱动的概念，方便用户扩展自定义的集群驱动，以驱动的方式提供了对不同容器集群的支持。Magnum 驱动是对服务器类型

(虚拟机或物理机)、操作系统类型、容器编排引擎 (COE) 的不同组合的抽象。

Magnum 驱动通过 `stevedore` 来实现动态配置和扩展, `entry_point` 定义在 `setup.cfg` 的 `magnum_drivers` 字段。同时 Magnum 提供了一个命令行工具 `magnum-driver-manage` 用于查看目前加载的驱动列表。

Magnum 驱动的代码位于 `magnum/drivers/` 目录, 每一个驱动都是单独的一个目录, 命名方式为 “`coe_os(_server_type)_版本号`”。

```
.
├── common
├── __init__.py
├── k8s_coreos_v1
├── k8s_fedora_atomic_v1
├── k8s_fedora_ironic_v1
├── mesos_ubuntu_v1
└── swarm_fedora_atomic_v1
```

每一个 Cluster 驱动都包含类似的目录结构:

```
.
├── driver.py
├── __init__.py
├── template_def.py
├── templates
│   ├── COPYING
│   ├── fragments
│   ├── kubeclasser.yaml
│   ├── kubemaster.yaml
│   └── kubeminiion.yaml
└── version.py
```

`driver.py` 继承自 `common/driver.py` 并定义了该驱动的 `providers`:

```
class Driver(driver.Driver):
    provides = [
        {'server_type': 'vm',
         'os': 'fedora-atomic',
         'coe': 'kubernetes'},
    ]

    def get_template_definition(self):
        return template_def.AtomicK8sTemplateDefinition()
```

`get_template_definition` 方法会返回该驱动对应的 Heat 模板文件的存放位置。

```
class AtomicK8sTemplateDefinition(kftd.K8sFedoraTemplateDefinition):
    """Kubernetes template for a Fedora Atomic VM."""
```



```

@property
def driver_module_path(self):
    return __name__[:__name__.rindex('.')]

@property
def template_path(self):
    return os.path.join(os.path.dirname(os.path.realpath(__file__)),
                        'templates/kubecoluster.yaml')

```

- `*cluster.yaml` 是该 driver 下的 cluster stack 的 Heat 模板定义。
- `*master.yaml`、`*minion.yaml` 是 `*cluster.yaml` 的子资源，分别定义了管理节点和从节点的资源配置情况。
- `fragments` 目录下包含的是一些 shell 脚本文件或者 yaml 文件，用于执行特定的集群配置。
- 一些通用的配置脚本文件放在 `driver/common` 目录中。

如果用户需要新增加额外的驱动，可仿照 `magnum/drivers` 目录中已有的驱动模板，使用 `driver/common` 目录下的虚拟机配置脚本来添加新的 heat 模板文件。

6. 制作启动镜像

无论使用 Magnum 部署基于虚拟机的容器集群还是物理机的容器集群，都需要使用一个操作系统镜像，Magnum 代码库中提供了不同镜像的制作脚本。Magnum 使用 `diskimage-builder`（OpenStack 的一个项目，提供云环境下各种操作系统的定制）工具制作 Magnum 创建的容器集群运行的操作系统镜像。目前 Magnum 支持 `fedora-atomic`、`ubuntu`、`coreos` 等常见的用于运行 Docker 容器的宿主系统。

Magnum 项目创建已经快两年了，从最初的快速发展到 2016 年奥斯汀峰会的项目目标变更之后，更明确了 Magnum 项目的重点：聚焦容器集群引擎的编排，并逐步加强扩展性、并发性，以及自动扩容/缩容的等集群的管理的高级功能。

Magnum Mitaka 版本加强了 `ironic driver` 的支持，也就是说使用 Magnum 在 OpenStack 集群中我们可以在裸机或者虚拟机上部署和运行容器技术。

12.2.3 Murano

Murano 提供应用程序目录服务。第三方应用开发者与管理员可以利用 Murano 快速发布各种云应用，同时用户（包括没有任何经验用户）能够通过 Murano dashboard 挑选出所需要的应用以及相关部件，并“一键”傻瓜式部署该应用到云环境中。

Murano 项目包含多个源码仓库。

- `murano`：主要代码仓库，包含 Murano API、Murano engine 和 MuranoPL。

- murano-dashboard: 通过 Horizon 插件机制提供 Murano 面板。
- murano-agent: 运行在客户虚拟机中的代理, 负责在虚拟机中部署相应的应用程序。
- python-muranoclient: Murano 的 Python 客户端程序。
- murano-apps: 应用程序仓库提供各种已发布的应用给用户。

1. Murano 体系结构

Murano 体系结构如图 12-7 所示。

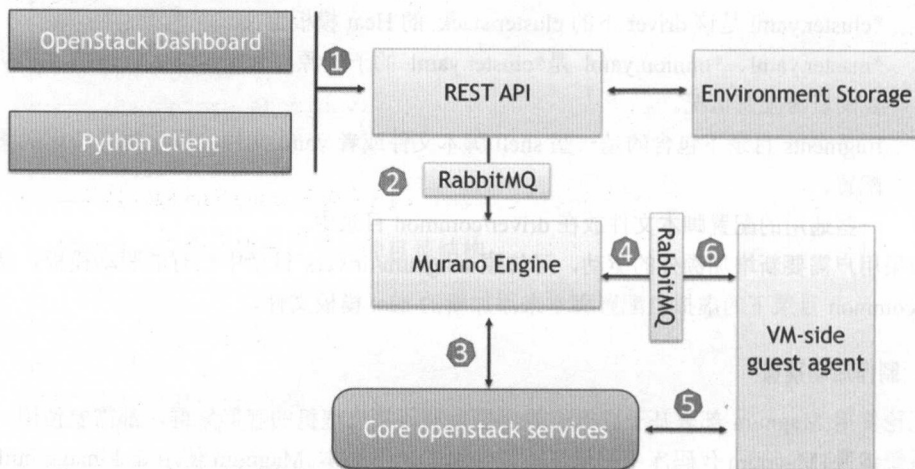


图 12-7 Murano 体系结构

- API Service: 向外提供 Restful API。
- Murano Engine: Murano 核心组件, 接收并解析 API Service 发来的各种 request (包括创建/更新/删除应用) 后, 创建相应的 Heat Template 来分配应用所必需的资源, 并准备 Execution Plan 提供给 Murano Agent 在虚拟机中部署应用。
- Murano Agent: 安装在虚拟机中的代理程序, 接收并执行 Murano engine 发来的 Execution Plan 部署应用。

用户通过 Murano dashboard 挑选并部署应用程序后, Murano 的工作流程如图 12-7 所示。

1) 用户通过 Murano dashboard 或者 python-muranoclient 发出请求到 Murano API, 要求部署相关应用程序到某一工作环境中。

2) API Service 接收到用户请求后, 在后台数据库进行相应操作后, 将请求通过 RabbitMQ 队列发给 Murano Engine。

3) Murano Engine 从 RabbitMQ 队列中取出该请求并解析后, 根据该应用所需要的资源创建 Heat Template, 通过 Heat 创建所需底层资源。

4) Murano Engine 根据部署的应用程序生成对应的 Execution plan, 放入 RabbitMQ 队列

中等待虚拟机中的 Murano Agent 提取用来部署应用。

5) 通过 Heat 创建对应的虚拟机以及其他底层资源，包括网络以及路由等。

6) 该虚拟机镜像中预置了 Murano Agent，在虚拟机启动后，Agent 会自动从 RabbitMQ 队列中取出相应的 Execution Plan 并执行来部署用户所需的应用程序。在完成部署后，Agent 会通知用户该工作环境已经部署完成。

2. Murano 应用商店

Murano 提供了官方应用商店 <http://apps.openstack.org/>，如图 12-8 所示，用户以及管理员可以从下载所需要的应用程序到自己的云平台中。该应用商店提供了 3 种类型的内容提供下载。

- **Murano packages:** 提供 Murano 应用程序包格式的多种应用程序，包括单节点以及集群多节点的多个版本。
- **Heat Templates:** 将已有的多个 Heat Template 包装成 Murano Packages 格式。
- **Glance Images:** 针对 Murano 定制后的各种镜像，其中包括已预置了 Murano Agent 的镜像。

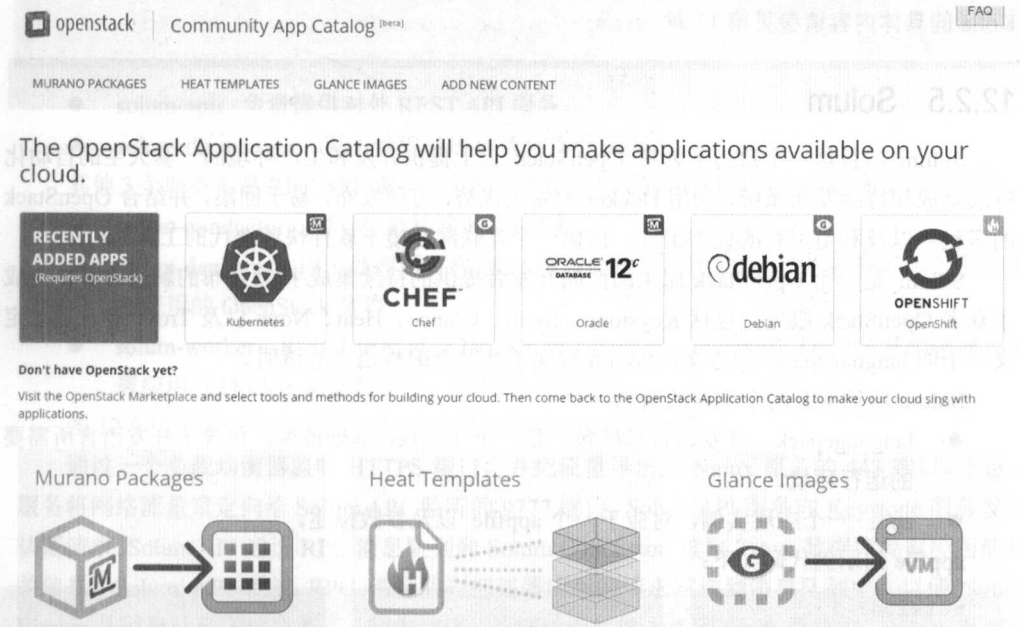


图 12-8 Murano 应用商店

3. Murano 应用程序包格式

Murano 应用程序软件包的结构是一个预先定义并压缩的 ZIP 格式的应用程序目录。用户

能够将其上传至 Murano 应用商店。该根目录包括以下主要的结构。

- `manifest.yaml` 文件：应用程序的入口，且文件名是固定的，不可以使用任何其他名称。
- `Classes` 文件夹：包含了 `MuranoPL` 类的定义，用于定义 `Murano` 生成该应用程序 `Murano Engine` 所需要执行的流程。
- `Resrouces` 文件夹：包含了可执行的计划模板，以及 `scripts` 文件夹，还包含了 `Murano agent` 部署应用所需要的文件。
- `UI` 文件夹：包含了动态的 `UI.yaml` 定义。动态 `UI` 的主要目的是为了生成应用时动态的创建表单。`Murano` 控制面板对于应用程序没有任何的了解，所以所有的应用定义须包含一个说明，告诉控制面板如何去创建一个应用。
- `logo.png` 文件（可选）：该应用程序所分配到的图标文件。
- `images.lst` 文件（可选）：包含由该应用程序所需的镜像列表。

12.2.4 Kolla

Kolla 项目于 2014 年 9 月成立，聚焦于如何使用 `Docker` 容器部署 `OpenStack` 服务。有关 Kolla 的具体内容请参见第 13 章。

12.2.5 Solum

Solum 项目是一个致力于为在 `OpenStack` 云上提供开发和生产环境的一套天生的自动化持续集成和持续发布系统，利用 `Docker` 的先天优势，方便发布，易于回滚，并结合 `OpenStack` 的多租户以及利用现有的优秀组件，提供一个互联网环境下软件快速迭代的工具。

Solum 是一个 `OpenStack` 原生的面向开发者提供的持续集成/持续发布的解决方案，集成了众多 `OpenStack` 服务，包括 `Keystone`、`Swift`、`Glance`、`Heat`、`Nova` 以及 `Trove` 等，通过定义一个叫 `languagepack` 的方案，Solum 实现了开发者编程语言的透明。

Solum 项目中有两个重要的概念。

- `languagepack`：开发语言基础包，是一个 `Docker` 基础镜像，包含了开发语言所需要的运行环境。
- `app`：一个应用实例，对应于一个 `appfile` 以及参数描述。

`appfile` 的示例代码如下：

```
version: 1
name: cherrypy
description: python web app
languagepack: python
source:
  repository:
    https://github.com/rackspace-solum-samples/solum-python-sample-app.git
```

```

revision: master
workflow_config:
  test_cmd: ./unit_tests.sh
  run_cmd: python app.py
trigger_actions:
- test
- build
- deploy
ports:
- 80

```

从中我们可以看出 `appfile.yaml` 定义了该 App 的描述信息，使用到的 `languagepack`，源代码目录等以及该 App 支持的工作流程。

从 Solum 项目源码的 `setup.cfg` 文件我们可以得知 Solum 的一些基本组件。

```

[entry_points]
console_scripts =
    solum-api = solum.cmd.api:main
    solum-conductor = solum.cmd.conductor:main
    solum-db-manage = solum.cmd.db_manage:main
    solum-deployer = solum.cmd.deployer:main
    solum-worker = solum.cmd.worker:main

```

- `solum-api`: 负责提供对外 REST API 服务。
- `solum-db-manage`: 用于管理员进行数据库升级操作。

其他 3 个服务都是 RPC API 服务,用于内部组件之间的相互调用,响应内部事件的处理。

- `solum-conductor`: 用于处理构建 build 任务。
- `solum-deployer`: 用于处理所有和 Heat 服务的接口,比如创建/更新/删除用于部署新的应用的 OpenStack 集群。
- `solum-worker`: 用于处理所有与 Docker 相关的操作,比如制作 Docker 基础镜像,部署应用到 Docker 基础镜像中,执行单元测试等。

图 12-9 所示为一个典型的在生产环境部署 Solum 的架构图。

通过一个负载均衡器监听 HTTPS 端口,并把流量导出到 Nginx 服务的 443 端口。Nginx 服务将网络流量重定向给 Solum API 监听的 9777 端口。Solum API 服务向 Keystone 服务发起认证请求。Solum API 通过 RPC 消息队列向 Solum Conductor 读取 Trove 数据库提取应用的相关信息。Solum API 通过 RPC 消息队列把部署应用的请求发送到消息队列中,以便 Solum Worker 从消息队列中读取消息,制作镜像,并把镜像文件上传到 Swift 服务中。Solum Worker 通过 Solum Conductor 服务将应用到状态信息持久化到 Trove 数据库中。Solum 从消息队列中获取部署应用的请求,调用 Heat 服务在 OpenStack 集群中申请资源用于部署应用,然后将应用的状态信息经 Solum Conductor 持久化到 Trove 数据库中。

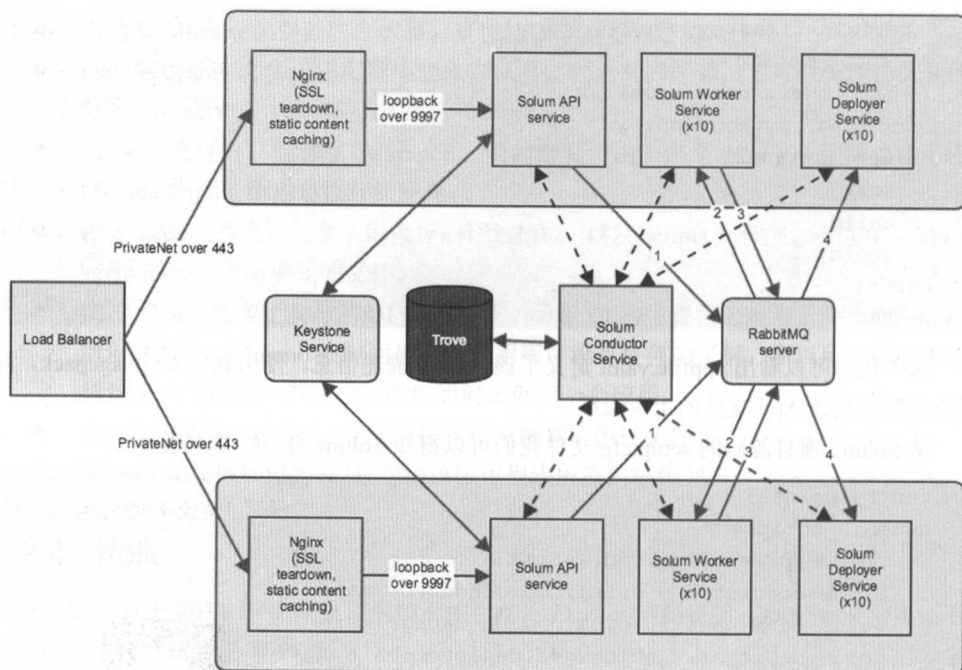


图 12-9 Solum 架构

12.2.6 Kuryr

试想一下如果在 OpenStack 集群中创建一些 Docker 主机，其中一台 Docker 主机上的 container 和另一台 Docker 主机上的 container 之间的网络是如何连接的呢？

如图 12-10 所示，我们需要在 Neutron Overlay 网络上再创建一层 Flannel Overlay 的网络，这种 Overlay 网络的性能会有明显的下降。

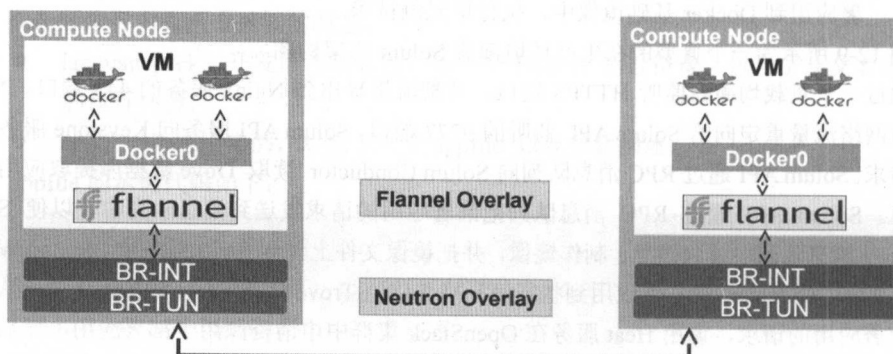


图 12-10 传统 Docker 网络架构

如果 Docker 可以直接使用 Neutron 提供的网络,那么网络性能将得到极大的改善。Neutron 本身就是生产级的 OpenStack 网络解决方案,后端支持多种商用网络解决方案,且 Neutron 提供了更高级的策略控制,多租户等特性。Docker 使用 Neutron 提供的网络将获得更多的特性支持。

为了把容器和 Docker 网络加入到 Neutron 解决方案和网络服务的使用中,Kuryr 项目应运而生。Kuryr 一词来自捷克语,意思是“信使”。Kuryr 旨在为 Docker 提供一个 OpenStack 的远程网络服务,成为一座连接 Docker 和 Neutron 两个社区的“整合桥梁”。简单来说,就是基于 Kuryr,Docker 的网络可以由 OpenStack 的网络服务 Neutron 管理。

目前 Kuryr 项目由 Kuryr 自身以及两个子项目 Kuryr-libnetwork 和 kuryr-kubernetes 组成,Kuryr 包含了通用的 lib 库,kuryr-libnetwork 目的是为 Docker 提供基于 Neutron 的 Docker 网络驱动,kuryr-kubernetes 目的是为了 Kubernetes 集群提供基于 Neutron 网络服务。

Docker 自 1.9 版本开始分离出了 libnetwork,使得 Docker 支持使用第三方的网络驱动。如图 12-11 所示,Kuryr 实现了一个 Docker 的网络驱动,并实现了 Docker 的容器网络模型(CNM),把所有 Docker 跟网络相关的请求转发给 Neutron 服务。

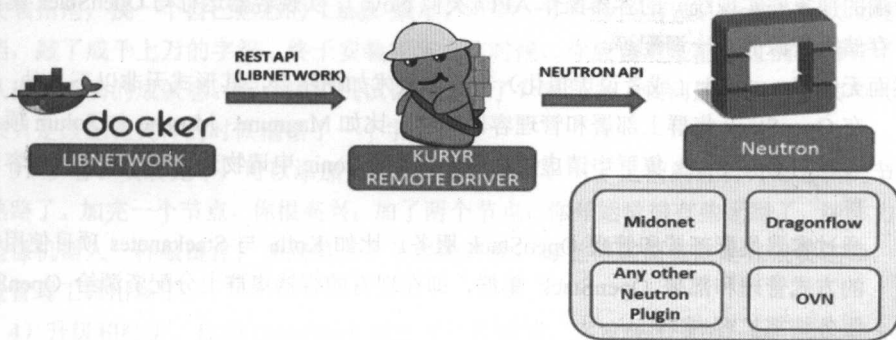


图 12-11 Kuryr 架构

Neutron 接到请求后根据自身后端网络驱动为 Docker Container 创建相应的网络,端口以及相关的访问策略等。

Kuryr 就像一个信使一样,把 Neutron 提供的网络,端口信息返回给 Docker,Docker 并不需要感知谁提供的网络设备。除此之外,Kuryr 还需要根据 Neutron Port 的类型提供 Container 的 Namespace 端口与 Neutron Port 的绑定操作。

Kuryr 实质上是一个本地 REST API Server,所以全部处理逻辑通过 REST API Controller 就可以实现。它提供了 IP 地址管理以及二层网络以及端口管理等功能。

OpenStack Neutron 中的某些网络概念与 Docker 网络模型中的概念有一些小的区别,比如 Neutron 中的 Port,在 Docker 网络模型中叫做 end_point。

对于希望了解 Kuryr 如何做到网络操作请求转发的读者可以参见

`kuryr_libnetwork/controllers.py` 中的代码。基本上就是按照 Docker libnetwork remote driver 的实现 (<https://github.com/docker/libnetwork/blob/master/docs/remote.md>)。

额外的工作是在 Kuryr Server 启动的时候, 创建了一个 Neutron Client 实例用于与交互 Neutron Server 交互。

Kuryr 转发 Docker libnetwork 的 Remote API 请求到 Neutron, 通过 Neutron Server 最终把请求传递给 Neutron Agent 创建一个供 Docker 容器使用的 Port 之后, 如果 Docker 容器希望使用这个 Port, 需要做一个 bind 操作。简单地说就是把处于 Docker 容器网络命名空间中的网络接口与 Neutron Agent 创建的网络接口连通。

12.2.7 容器技术与 OpenStack 的展望

容器技术与 OpenStack 社区的快速发展, 也使得人们对 Pass 和 IaaS 有了更新的认识。开源社区的工程师们尝试不同的解决方案能够让容器和虚拟化技术更好地结合, 并为生产生活提供更易用的解决方案。至本书完成之时, 社区中还有一些新创建的优秀项目在孵化中。比如 Stackanetes 项目使用 Kubernetes 部署和管理 OpenStack 服务, Zun 项目致力于提供各种容器技术后端的抽象并实现统一的容器操作 API (类似 Nova), 实现容器运行与 OpenStack 集群中网络、存储等资源的统一调配等。

然而无论 OpenStack (或者说虚拟化) 与容器技术如何结合, 其形式无非以下 3 种:

- 在 OpenStack 集群上部署和管理容器集群。比如 Magnum、Murano 与 Solum 项目, 通过向 OpenStack 集群申请虚拟资源 (或通过 Ironic 申请物理资源) 部署一套容器集群。
- 通过容器集群部署和管理 OpenStack 服务。比如 Kolla 与 Stackanetes 项目使用容器的方式管理和部署 OpenStack 集群, 即在现有的容器集群上分配资源给 OpenStack 服务按需扩容/缩容。
- OpenStack 集群与容器集群同时存在于整个物理集群上。

硬性地将二者结合不是目的, 我们的目的是能够让资源分配更高效, 资源形式更灵活, 资源之间的连接更可控。

随着 OpenStack 各个组件的逐渐稳定成熟，面对众多的 Service，繁琐的配置选项，如何方便地部署和运维成了一个热点话题，也是整个 OpenStack 生态中不可缺少的一环。对于开发人员来说，最理想的环境当然是 Devstack，不仅可以一键部署，而且方便调试，但是对于生产环境来说，情况要复杂得多。

一个懵懂的运维人员成长为 OpenStack 熟练工，大约需要这样一个过程。

1) 泛读。翻看各种 OpenStack 文档，包括官方文档、名家博客，以及他人的学习笔记，先理解 Nova 是什么，Swift 是做什么的，Keystone 什么原理，Neutron 的功能设计，等等，看一圈下来，回头想一想，还是云里雾里，不明白这 OpenStack 究竟是什么。

2) 安装。虽然没有完全掌握，还是得硬着头皮上。找来官方文档 docs.OpenStack.org 上的安装指南，挑一个自己熟悉的 Linux 版本，开始一步一步照着做。经过若干次反复地查阅文档，敲了成千上万的字符，终于安装完毕。这时候，你应该对之前学过的东西有了进一步的认识，满满的成就感。起个虚拟机试试，出错了？看日志，找问题，折腾了大半天，发现原来只是改配置文件的时候输错了一个字……

3) 扩容。安装完毕，可以添加更多的计算节点了。有了之前的经验，部署新的节点就轻车熟路了。加完一个节点，你很高兴；加了两个节点，你开始觉得有些无聊了，纯体力劳动，只是像机器人一样敲键盘；加到第三个节点的时候，你觉得这简直就是在浪费生命，于是该配置管理工具出场了。

4) 升级和维护。你的 OpenStack 运行了一段时间，你发现了一些 bug，而这些 bug 在新版本已经解决了；你希望用到某些新功能，而新功能只在新版本中才有；你希望更改一些配置，以更好适应你的需求。所有这些工作如果没有得心应手的工具，就是一场无休无止的运维噩梦。

于是，你开始或者用脚本，或者用配置管理工具，把上面的任务自动化。这就是 OpenStack 各个部署项目的雏形。

每一个 OpenStack 部署项目，后面都站着一家或多家 OpenStack 厂商，作为自己云方案中的部署工具，不仅方便了服务人员，更凝聚着多年使用中累计的经验，为后来者提供了非常好的学习素材。

13.1 配置管理工具

工欲善其事，必先利其器。通过自动化工具来完成系统部署、系统参数配置、软件安装等一系列运维工作，已经是系统管理员的必备技能。目前流行的配置管理工具主要有 Puppet、Chef、Ansible 和 Salt 等。

关于如何选择这些工具，是个仁者见仁，智者见智的事情。过去的经验及技术背景会是一个重要的影响方面，每个人都倾向于选择自己熟悉的工具。好在这些工具都可以满足正常的工作需要，又各有特点，如图 13-1 所示，所以不需要在这个上面浪费太多的时间。

	语言	发布时间	模块扩展	无代理执行	图形界面
puppet	Ruby	2005	支持	不支持	支持
chef	Ruby	2009	支持	不支持	支持
ansible	Python	2012	支持	支持	支持
salt	Python	2011	支持	支持	支持

图 13-1 主要配置管理工具比较

Puppet 是最早和最成熟的，于 2005 年发布，用 Ruby 编写，扩展模块需要用 Puppet 自己定义的描述性语言，或者 Ruby DSL，对之前不熟悉 Ruby 语言的人有一定的门槛。

Chef 也是用 Ruby 语言编写的，第一个版本发布于 2009 年。

Ansible 发布于 2012 年，用 Python 编写。简单易用是 Ansible 最大的特点，扩展模块可以用各种语言包括脚本，只要符合接口定义即可。Ansible 不需要在被部署机器上装 Agent，利用 ssh 来执行命令和复制模块到目标机执行。当然，也由于这种简单的结构，导致 Ansible 在部署规模变大的时候性能会有些力不从心。

Salt 于 2011 年发布第一个版本，与 Ansible 一样用 Python 来编写。Salt 的扩展模块可以用 Python 或者是 pyDSL，对于 OpenStack 开发者来说不是问题。Salt 支持像 Ansible 一样无 Agent 执行，也可以用 Master-Minion（服务端/客户端）模式。Salt 可以支持级联的 Master，使它在扩展性上大大优于其他的系统。

Ansible 和 Salt 是新型配置管理工具的代表，在有些功能上，比如图形界面方面还有些欠缺，但是复杂性比 Puppet 和 Chef 低很多，容易入门，在 Operator 中赢得了很好的口碑和很多新用户。

这里只对 Ansible 进行介绍。

Ansible 要求有一台安装了 Ansible 软件的机器作为管理节点，可以通过 ssh 访问到被管理的机器。对于被管理的机器只需要安装了 Python 运行环境即可。这里通过一个最简单的例子演示一下 Ansible 的基本用法和概念，Ansible 的网站有详细的官方文档可供查阅（<http://docs.ansible.com/>）。

(1) 安装

一般 Linux 发行版都带有 Ansible 的安装包，可以直接安装，以 Ubuntu 为例：

```
$ sudo apt install ansible
```

但是一般自带的版本比较老，Ansible 又是个非常活跃的社区，所以通常用 pip 来安装最新的稳定版本：

```
$ sudo pip install ansible
```

(2) inventory 文件

inventory 文件的目的是告诉 Ansible 管理哪些机器。一般在命令行中用“-i”选项指定该文件的位置。如果没有指定，则 Ansible 会使用默认的/etc/ansible/hosts。下面是一个最简单的 inventory 文件的例子，指定了一个主机 192.168.0.149，同时指定它的分组是 webserver。

```
$ cat hosts
[webserver]
192.168.0.149
```

执行第一个命令：

```
$ ansible -i hosts webserver -m ping
192.168.0.149 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

对 webserver 这个组的主机执行 ping 操作，这里的 ping 不是 Linux 命令，而是 Ansible 的一个模块，用来检查目标机是否可用。

(3) 模块

Ansible 对被管理机器所执行的操作都是由模块完成的，模块就相当于 Ansible 的命令。“-m”参数用以指定模块，“-a”参数用来传递模块的参数。对于用户需要的大多数操作，都有已经实现的核心模块，可以直接使用。对于一些定制化的需求，用户可以开发自己的模块来实现。所有 Ansible 内置的核心模块都有对应的使用帮助：

```
$ ansible-doc -list //列出所有核心模块
$ ansible-doc lineinfile //查看 lineinfile 模块的帮助
```

下面就以 lineinfile 模块为例做个简单介绍：

```
$ ansible -i hosts webserver -m lineinfile -a "dest=/home/ubuntu/hello.txt
line='Hello World'"
192.168.0.149 | SUCCESS => {
    "backup": "",
    "changed": true,
    "msg": "line added"
}
```


这个命令是确保 webserver 组的所有主机上，目标文件里都包含 Hello World 这一行。在我们执行的时候 hello.txt 是个空文件，所以，返回结果显示 line added。

我们再执行一遍这个命令：

```
$ ansible -i hosts webserver -m lineinfile -a "dest=/home/ubuntu/hello.txt
line='Hello World'"
192.168.0.149 | SUCCESS => {
    "backup": "",
    "changed": false,
    "msg": ""
}
```

可以发现返回结果依然是 SUCCESS，但是“changed: false”表示目标机器已经满足了要求，不需要改变。

Ansible 和其他配置管理工具一样，是确保目标机器到达一个命令中定义的状态，重复执行 Ansible 命令既不会执行不必要的操作，也不会损坏系统的状态。

(4) 编排 (Playbook)

以上都是直接用 Ansible 来执行单个命令做演示，复杂命令的执行则需要用到 ansible-playbook。在复杂的使用环境中，需要用到 Playbook 把需要的配置编排好，达到自动化的目的。Playbook 使用 YAML 格式，文件名以 yml 结尾。上面的功能用 Playbook 就是这样的：

```
$ cat hello.yml
---
- hosts: webserver
  tasks:
    - name: Ensure "Hello World" is in the line
      lineinfile: dest=/home/ubuntu/hello.txt line='Hello World'
$ ansible-playbook -i hosts hello.yml
.....
192.168.0.149 : ok=2    changed=1    unreachable=0    failed=0
```

以上仅是对 Ansible 做一些基本说明，便于我们后续对 OpenStack 部署项目的理解。后面我们会看到，Ansible 有很多高级的特性，可以对 host 按照功能/位置分组，定义不同的变量，用 role 来定义不同的配置任务，模板，等等，足够完成像 OpenStack 部署这样复杂的任务。

13.2 OpenStack 部署项目

OpenStack 的部署一般包括两个部分的工作：基础操作系统的提供与 OpenStack 组件的安装。

基础操作系统的提供，传统上是通过 Cobbler（一个用来部署和安装系统的开源项目），搭建一个 PXE 的安装环境，通过 kickstart 文件实现全自动安装和简单设置。这种方式节约了人工，但安装时间比较长，系统配置也不够灵活。目前新兴的解决方案是预先制作好磁盘镜像，用 PXE 启动到一个预先定制的系统，然后把准备部署的镜像 dd 到目标磁盘，再通过 cloud-init 来完成系统的定制化。这种方式可以在几分钟内完成一个系统的安装，OpenStack 的 Ironic 项目，以及 Ubuntu 的 MAAS 都是通过这种方式管理部署系统的。

而 OpenStack 组件的部署就要借助于 Puppet/Ansible 这样的工具，每一个配置管理工具在 OpenStack 中都有对应的项目，比如 puppet-OpenStack、OpenStack-Chef、OpenStack-Ansible 和 OpenStack-Salt 等。这些项目都可以方便地进行生产环境多节点 OpenStack 的部署，也是其他集成部署项目的基础。

在解决了初始的部署问题后，“Day 2 Operations”的问题就浮出了水面，包括 OpenStack 的扩展、升级、配置更新等，这是传统的部署项目所欠缺的，而新兴的 OpenStack 部署项目 Kolla 则充分利用了容器技术，致力于降低 OpenStack 部署维护的复杂度，给用户提供一个固化了社区经验的部署工具。

13.2.1 Bifrost

Ironic 项目可以用于物理机的管理，因此它也可以作为 OpenStack 的原生项目来承担 OpenStack 部署中基础操作系统的部分。

TripleO 就是一个以 Ironic 为基础，充分利用 Neutron、Glance、Heat 等各个服务来进行 OpenStack 部署的社区项目。TripleO 要求先搭建一个底层云，并以此为基础定义和部署上层云，各个部分耦合紧密，关系比较复杂。它的目标是一个面向最终用户的解决方案，对初学者难度稍大。对于开发者来说，最喜欢的方案是简洁有效，易于学习，因此针对基础操作系统安装这个任务，Bifrost 无疑是一个更为轻量级的选择。

Bifrost 由一系列 Ansible 脚本组成，它可以在一套已知的硬件上通过 Ironic 自动部署一个基本的镜像，利用 Ironic 的 standalone 模式（不需要 Keystone 及 Nova 等其他 OpenStack 服务）来管理部署硬件机器。

如图 13-2 所示，Bifrost 以 Ironic 为基础，首先准备一个可以 PXE 安装的环境，然后注册节点的电源管理信息，提供给被管理节点安装操作系统的服务。

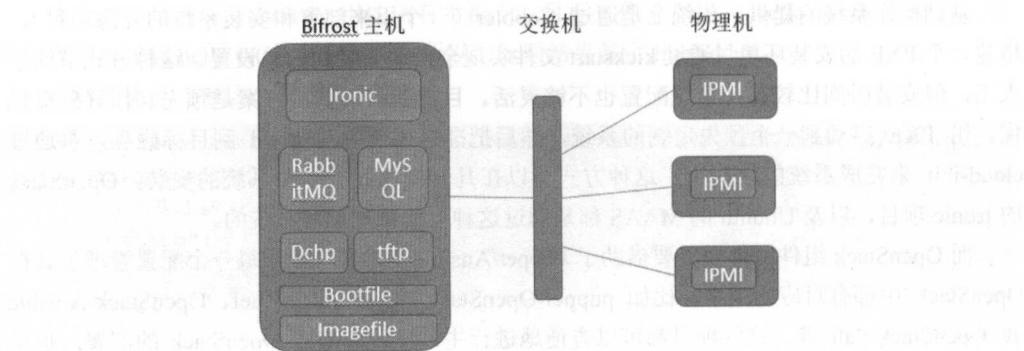


图 13-2 Bifrost 结构

Bifrost 主要完成 3 个阶段的工作：

1) 安装。主机环境准备，安装和配置物理机部署所依赖的所有组件。

- Standalone Ironic。
- RabbitMQ。
- MySQL。
- TFTP 服务器。
- DHCP 服务器。
- 用 diskimage-builder 创建部署用的磁盘镜像。

在这个阶段开始之前，需要设置一些变量来定义你的环境，比如说 `network_interface` 表示要在哪个网卡上启用 DHCP 服务。浏览 `playbooks/inventory/group_vars/localhost` 文件可获得可配置的选项。

```
$ cd bifrost
$ bash ./scripts/env-setup.sh //完成环境初始化，安装 ansible
$ source ${ANSIBLE_INSTALL_ROOT}/ansible/hacking/env-setup //添加 Ansible
到执行路径中
$ ansible-playbook -i inventory/localhost install.yaml //执行安装任务
```

2) 注册。动态硬件注册，把需要部署的目标机信息注册到 Ironic 数据库中。需要包含的目标机硬件信息有以下内容。

- MAC Address: 表示目标机的 MAC 地址。
- Management username: 表示 IPMI 接口的用户名。
- Management password: 表示 IPMI 的密码。
- Management Address: 表示 IPMI 的地址。
- CPU Count: 表示 CPU 的数量。
- Memory size in MB: 表示内存数量。
- Disk Storage in GB: 表示磁盘大小。

- Host or Node name: 表示部署完成后设置的主机名。
- Host IP Address to be set: 表示部署完成后设置的 IP 地址。
- ironic driver: 使用的 Ironic 驱动, 比如 agent_ipmi。

目标机的硬件信息可以用 CSV/JSON/YML 文件来提供, 在 `playbooks/inventory/` 文件夹下面分别有针对不同格式的例子, 只要把信息替换成自己的硬件即可。编辑完成硬件信息文件之后运行下面命令, 即可以把硬件信息注册进去:

```
export BIFROST_INVENTORY_SOURCE=/tmp/baremetal.json
ansible-playbook -i inventory/bifrost_inventory.py enroll-dynamic.yaml
```

运行 “`source env_vars; ironic node-list`” 就可以看到刚刚注册的节点。

3) 部署。利用 Ironic 将操作系统部署到每个目标机上。

- 配置 pxe 环境。
- 用 IMPI 来设置目标机从网络启动。
- 启动物理机到 RAMdisk, 并且部署磁盘镜像到硬盘。
- 重新启动目标机, 用 cloud-init 来初始化系统。

完成节点注册后的部署工作主要由 Ironic 来驱动完成, 运行如下命令:

```
export BIFROST_INVENTORY_SOURCE=/tmp/baremetal.json
ansible-playbook -i inventory/bifrost_inventory.py redeploy-dynamic.yaml
```

可以用 “`ironic node-list`” 命令来查看节点的状态。

Bifrost 抽象了物理机的安装过程, 只需要用户以尽量简单的方式提供必需的信息, 分 3 个步骤/阶段, 把物理机操作系统的安装做到了自动化。

13.2.2 Kolla

Kolla 最早于 2014 年提出的。刚一提出便得到了广泛的支持, 并很快成为了 OpenStack 正式项目。从 Liberty 版本开始, Kolla 的 commit 数目一直在前 10 名之内, 且参与厂商众多。

两年的时间里, Kolla 经历了从单机 demo 到 Ansible 多机部署, 从单纯的部署到升级、重新配置支持, 逐渐完善至产品级别。最近的动态是跟容器编排工具的进一步结合, 试图把 OpenStack 集群放入 Kubernetes 管理。Stackanetes 和 Kolla-kubernetes 是这方面探索的两个项目。有趣的是, Kolla 最早的提议便是以 Kubernetes 来管理, 可惜当时各方面条件尚未成熟, 两年之后, 社区又重新开始考虑两者的结合。

Kolla 的目标有两个, 一是提供生产环境可用的容器镜像, 二是提供部署工具可以快速的部署方便的管理。

如图 13-3 所示, 首先 Kolla build 根据 Dockerfile 来制作镜像, 并放入本地的镜像仓库(Local Docker Registry) 中, Kolla Deploy 会根据用户配置, 自动生成配置文件, 把配置文件和镜像仓库中的镜像一起部署到对应的节点上。

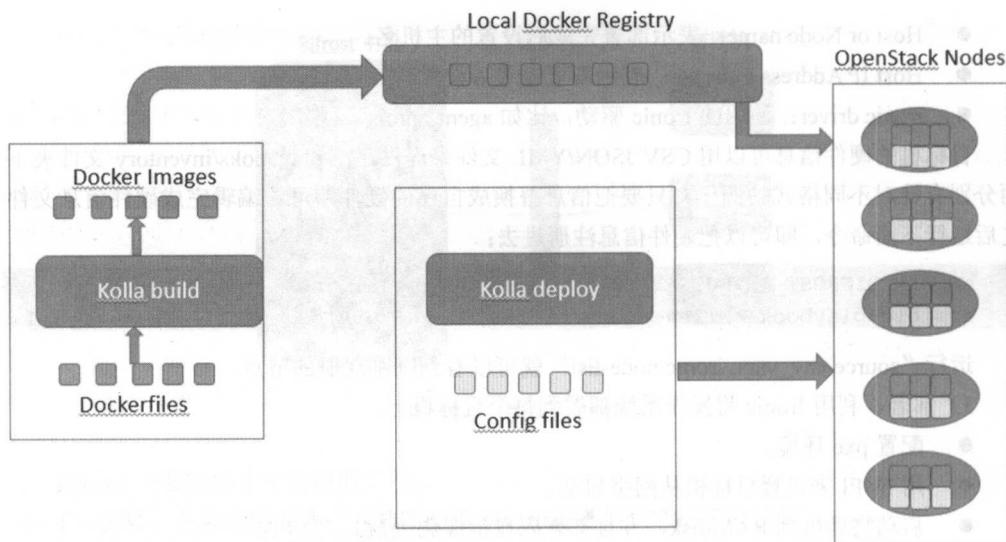


图 13-3 Kolla 工作流程

Kolla 的源码目录如下：

```

├── ansible - 用于部署，升级的 Ansible Playbook
├── demos
├── dev
├── doc
├── docker - 各个 image 的 Dockerfile 定义
├── etc - 配置文件
├── kolla - kolla 命令行的实现
├── releasenotes
├── specs
├── tests
└── tools - 辅助工具，以及 kolla-ansible
  
```

Kolla 主要的代码是 Dockerfile 和 Ansible 的 Playbook，是社区经验的集合，其他的文件都是做一些辅助性的工作，早期几乎都是用 bash 脚本写成的，后来为了项目管理更方便，一些命令改为用 Python 开发。

etc 目录放置 Kolla 需要的配置文件，kolla-build.conf 是制作镜像时使用的参数，globals.yml 是部署时使用的参数，passwords.yml 则用于定义部署 OpenStack 时用到的密码文件。

kolla 目录是一些 Python 实现的 Kolla 命令，目前包括 kolla-build 和 kolla-genpwd。

tools 是一些脚本工具的集合。kolla-ansible 是执行部署的命令，现在依然是用脚本实现。cleanup-containers、cleanup-host、cleanupimages 用于清理 Kolla 产生的容器及镜像，init-runonce 是部署完成后用来做初始化的，创建 Glance 镜像、网络等。另外有些脚本是开发者为了方便

开发和试验环境部署所做的一些工具，但是由于 Kolla 支持操作系统版本众多且变化很快，有些脚本可能需要修改才能使用。

docker 目录下是 Dockerfile，按照项目来划分，各个项目之间相对独立，而项目内部又根据复杂程度，划分成一个或者多个 service，每个 service 对应一个 docker image。

ansible 目录存放部署用的 Ansible Playbook。按照不同的项目和逻辑功能分成了不同的 role 来管理。Kolla 还对 Ansible 做了两个模块扩充，一个是 kolla_docker 实现对容器的管理；一个是 merge_configs，产生和管理 OpenStack 的配置文件。

1. Docker 镜像

我们首先看一下 Kolla 是如何定义镜像文件的。docker.base 是所有其他镜像的基础，下面是其 Dockerfile.j2 的代码片段：

```
FROM {{ base_image }}:{{ base_distro_tag }}
MAINTAINER {{ maintainer }}

LABEL kolla_version="{{ kolla_version }}"

{% import "macros.j2" as macros with context %}
{% block base_header %}{% endblock %}
{% include_header %}

ENV KOLLA_BASE_DISTRO {{ base_distro }}
ENV KOLLA_INSTALL_TYPE {{ install_type }}
ENV KOLLA_INSTALL_METATYPE {{ install_metatype }}

COPY kolla_bashrc /tmp/
RUN cat /tmp/kolla_bashrc >> /etc/skel/.bashrc \
    && cat /tmp/kolla_bashrc >> /root/.bashrc

ENV PS1="$(tput bold) $(printenv KOLLA_SERVICE_NAME) $(tput sgr0) $(id -un)@$(hostname -s) $(pwd)]$ "

{% if base_distro in ['fedora', 'centos', 'oraclelinux', 'rhel'] %}
... ..
{% endif for base_distro centos, fedora, oraclelinux, rhel #}
{% elif base_distro in ['ubuntu', 'debian'] %}
... ..
{% endif for base_distro ubuntu, debian #}
{% endif %}
```

可以看到，为了支持不同的 Linux 发行版，Kolla 采用了 jinja2 模板（一个现代的设计者友好地仿照 Django 模板的 Python 模板引擎），在正式 build 之前会用 kolla-build.conf 中定义的

变量代替。

Dockerfile 中主要的分支有以下几种：

- 操作系统类型 `base`，支持 `fedora`、`centos`、`oraclelinux`、`rhel`、`ubuntu` 和 `debian`，默认为 `centos`。
- 操作系统版本 `base_tag`，指定操作系统的版本，默认为 `latest`。
- 安装类型 `install_type`，指定安装类型，支持 `binary` 和 `source`，默认为 `binary`。

由于支持选项较多，导致 `Dockerfile.j2` 充满了 `if/else`，可读性较差，这里我们选择其中的一个分支 `ubuntu:latest`，`install from source` 来说明。

```
RUN if [ $(awk -F '=' '/DISTRIB_RELEASE/{print $2}' /etc/lsb-release) !=
"{{ supported_distro_release }}" ]; then \
    echo "Only supported {{ supported_distro_release }} release on
{{ base_distro }}" ; false; fi

# Customize PS1 bash shell
RUN cat /tmp/kolla_bashrc >> /etc/bash.bashrc

# This will prevent questions from being asked during the install
ENV DEBIAN_FRONTEND noninteractive

# Reducing disk footprint
COPY dpkg_reducing_disk_footprint
/etc/dpkg/dpkg.cfg.d/dpkg_reducing_disk_footprint

# Need apt-transport-https BEFORE we replace sources.list or apt-get update
wont work!
RUN apt-get update \
    && apt-get -y install --no-install-recommends apt-transport-https
ca-certificates \
    && apt-get clean

COPY sources.list.{{ base_distro }} /etc/apt/sources.list
COPY apt_preferences.{{ base_distro }} /etc/apt/preferences

{% set base_apt_packages = [
    'apt-utils',
    'curl',
    'gawk',
    'iproute2',
    'kmod',
    'lvm2',
    'open-iscsi',
    'python',
```

```

'sudo',
'tgt']
%}

.....

COPY set_configs.py /usr/local/bin/kolla_set_configs
COPY start.sh /usr/local/bin/kolla_start
COPY sudoers /etc/sudoers
COPY curlrc /root/.curlrc
RUN touch /usr/local/bin/kolla_extend_start \
    && chmod 755 /usr/local/bin/kolla_start
/usr/local/bin/kolla_extend_start /usr/local/bin/kolla_set_configs \
    && chmod 440 /etc/sudoers \
    && groupadd kolla \
    && mkdir -p /var/log/kolla \
    && chown :kolla /var/log/kolla \
    && chmod 2775 /var/log/kolla \
    && rm -f /tmp/kolla_bashrc \
    && curl -sSL
https://github.com/Yelp/dumb-init/releases/download/v1.1.3/dumb-init_1.1.3_
amd64 -o /usr/local/bin/dumb-init \
    && chmod +x /usr/local/bin/dumb-init

{% block base_footer %}{% endblock %}
CMD ["kolla_start"]

```

可以看到，base 镜像主要做的工作有：

- 基本检查和初始化环境。
- 设置 apt sources.list，由于要增加的 sources 比较多，所以，Kolla 把所有的改动都放在 sources.list.ubuntu 文件中，直接覆盖原来的文件。如果有本地的 apt mirror，开发者也可以设置自己的 apt 源。
- 安装基本包。
- 复制配置脚本 kolla_set_configs 和启动脚本 kolla_start。

这里需要说明的是，Kolla 生成的镜像是没有配置文件的，各个项目的配置文件需要在部署的时候传入。所以，Kolla 引入一个 kolla_set_configs 用来在启动容器的时候产生正确的配置文件。kolla_start 是容器的默认启动命令，源码位于 docker/base/start.sh：

```

#!/usr/local/bin/dumb-init /bin/bash
set -o errexit

# Wait for the log socket
if [[ ! "${SKIP_LOG_SETUP[@]}" && -e /var/lib/kolla/heka ]]; then

```

```

while [[ ! -S /var/lib/kolla/heka/log ]]; do
    sleep 1
done
fi

sudo -E kolla_set_configs
CMD=$(cat /run_command)
ARGS=""

if [[ ! "${!KOLLA_SKIP_EXTEND_START[@]}" ]]; then
    # Run additional commands if present
    . kolla_extend_start
fi

echo "Running command: '${CMD}${ARGS:+ $ARGS}'"
exec ${CMD} ${ARGS}

```

OpenStack 启动需要几十个容器，每个容器的启动方法都不相同，所以 Kolla 要求每个容器把自己的启动命令放在“/run_command”里，由 kolla_start 统一调用。如果该容器有需要额外的启动步骤，可以传入一个 kolla_extend_start 文件来完成。可以看到在 kolla_start 启动命令之前，会调用 kolla_set_config 来产生配置文件。

接下来是 openstack-base 容器 docker/openstack-base/Dockerfile.j2，基于之前的 base：

```
FROM {{ namespace }}/{{ image_prefix }}base:{{ tag }}
```

nova-base 容器 docker/nova/nova-base/Dockerfile.j2，基于 openstack-base：

```
FROM {{ namespace }}/{{ image_prefix }}openstack-base:{{ tag }}
```

Nova 其他的服务又基于 nova-base，docker/nova/nova-api/Dockerfile.j2：

```
FROM {{ namespace }}/{{ image_prefix }}nova-base:{{ tag }}
```

服务安装的过程跟普通的安装没有区别，只是替换为 Docker 的描述语言。

kolla-build 命令包装了 docker build，用于生成上述的镜像文件。用户既可以通过指定命令行参数的方式，也可以通过/etc/kolla/kolla-build.conf 配置文件来指定 build 选项。

2. Ansible Playbook

Kolla 以 Ansible 为部署工具，把之前生成好的 image 部署到目标机上，生成正确的配置文件，并且启动 service。下面我们来看一下 ansible 目录的结构。

首先按照项目，把针对每个项目要做的动作 role 来划分。比如 Nova，部署所需要的所有动作都在 ansible/roles/nova/tasks 目录下。

- config.yml: 生成 OpenStack 各个 service 需要的配置文件。
- bootstrap.yml: 对于一些容器，启动之前需要做一些初始化的工作，比如创建数据库、

用户名和密码等。

- start.yml: 启动容器。
- deploy.yml: config、bootstrap 和 start 的集合。
- reconfigure.yml: 配置发生变化时重新启动 service。
- upgrade.yml: 对 OpenStack 集群升级。

deploy/reconfigure/upgrade 是部署和维护一个 OpenStack 集群主要的任务。其他的角色 (role) 跟 Nova 类似, 不再赘述。

在浏览这些 yml 文件的时候, 我们注意到里面用到了大量的变量, 以适应不同的安装环境和配置。这些全局变量的定义在 ansible/group_vars/all.yml 文件中, 对于一些需要被管理员修改的变量放在了 etc/kolla/global.yml 中, 修改 global.yml 会覆盖原有的默认值。

下面的代码片段是启动 nova-api 与 nova-consoleauth 容器的例子, 使用了自己开发的 kolla_docker 模块, 指定启动的参数和挂载的卷。

```
# ansible/roles/nova/tasks/start_controllers.yml

---
- name: Starting nova-api container
  kolla_docker:
    action: "start_container"
    common_options: "{{ docker_common_options }}"
    image: "{{ nova_api_image_full }}"
    name: "nova_api"

    privileged: True
    volumes:
      -
        "{{ node_config_directory }}/nova-api/{{ container_config_directory }}:/ro"
        - "/etc/localtime:/etc/localtime:ro"
        - "/lib/modules:/lib/modules:ro"
        - "kolla_logs:/var/log/kolla/"
    when: inventory_hostname in groups['nova-api']

- name: Starting nova-consoleauth container
  kolla_docker:
    action: "start_container"
    common_options: "{{ docker_common_options }}"
    image: "{{ nova_consoleauth_image_full }}"
    name: "nova_consoleauth"
    volumes:
      -
        "{{ node_config_directory }}/nova-consoleauth/{{ container_config_directory }}:/ro"
```



```
- "/etc/localtime:/etc/localtime:ro"
- "kolla_logs:/var/log/kolla/"
when: inventory_hostname in groups['nova-consoleauth']
```

privileged: True, 由于很多容器是系统服务, 所以 Kolla 启动容器的时候采用了特权模式。Kolla 默认是在一个可信任的环境里运行, 并没有考虑安全隔离的问题。

“{{ node_config_directory }}/nova-api:{{ container_config_directory }}:/ro”, 表示默认把主机上的 /etc/kolla/nova-api 目录传入容器 /var/lib/kolla/config_files, 为每个 service 的配置文件。

“Kolla_logs:/var/log/kolla/”, 表示收集日志的数据卷。

when: inventory_hostname in groups['nova-api'], 当本机在 nova-api 组的时候, 才启动 nova-api 容器。对于 Nova 这个 role 来说, 需要支持所有的 Nova services, 但是这些 service 在部署的时候往往不在同一个主机上, 所以, Kolla 通过设置主机组来实现主机和 service 之间的灵活分配。被部署机器的分组在 ansible/inventory/multimode 中。

除了上述核心的功能外, 最新的 kolla-ansible 又加入了以下功能的支持。

- bootstrap-servers: 为被部署的机器准备运行环境, 包括 Docker 的安装, 以及一些环境的设置, 在 deploy 之前运行, 极大地方便了准备工作。
- deploy-bifrost/deploy-servers: 部署一个 Bifrost 容器, 并且自动实现对 server 的初始化安装。

Kolla 传统上要求被部署机器已经完成基础操作系统安装和配置, 集成了 Bifrost 以及系统初始化的设置后, Kolla 可以实现全套的从 Baremetal 到完成 OpenStack 部署, 再到后期运维的全生命周期管理。

kolla-ansible 是 Kolla 用于部署的命令, 包装了 ansible-playbook, 提供了一个抽象的接口, 便于用户调用不同的任务。具体请参考 <http://docs.OpenStack.org/developer/kolla/quickstart.html>。

13.2.3 TripleO

前面以两个社区项目 Bifrost 和 Kolla 为基础, 介绍了 OpenStack 自动化部署中需要解决的问题, 以及一种解决问题的方法。然而, 这两个项目都是开发者为导向的, 就是说, 注重于某一个阶段实际问题的解决, 而没有考虑最终用户应该怎么用。

接下来介绍的两个项目 TripleO 和 Fuel 是由 OpenStack 厂商主导的, 面向用户的 OpenStack 部署产品, 有着易于使用的界面和强大的管理功能。在 TripleO 和 Fuel 里, 不同的功能模块分成单独的项目开发管理, 最后组合成一个功能强大的产品。

TripleO 最为重要的一个特点是 V2P (Virtual Machine to Physical Machine)。在虚拟化的概念中, P2V 是一个比较常见的概念, 即把一个物理机上的内容做成镜像迁移到虚拟机中。而 V2P 则是一个逆过程, 是把虚拟机的镜像迁移到物理机上。

V2P 的好处显而易见, 部署虚拟机时之所以这么快, 是因为一切的安装配置过程已经在制作镜像模板的时候完成了, 只需要像挂载硬盘一样挂载上去就可以了, 由此引申到

OpenStack 部署当中，同样可以对不同类型的服务器节点事先制作好镜像，部署时像部署虚拟机一样，只要挂载镜像就行了，从而可以节省很多常规的安装配置时间。TripleO 就是采用了这种方式来完成部署。

TripleO 全称 OpenStack On OpenStack，顾名思义为“云上云”，可以简单理解为利用 OpenStack 来部署 OpenStack，即首先基于上述 V2P 的理念准备一个 OpenStack 节点的镜像，然后利用已有 OpenStack 环境的裸机服务（Bare Metal，也就是 Ironic 项目）去部署裸机，最后通过 Heat 项目再在裸机上配置运行 OpenStack。Redhat 最新的云平台 Redhat OpenStack Platform 9 就是基于 TripleO 技术构建的。

如图 13-4 所示，使用 TripleO 部署的 OpenStack 主要包括底层云（UnderCloud）与上层云（OverCloud）两个部分。底层云（部署云）包含部署和管理一个 OpenStack 云所需的组件，上层云（负载云）是客户的云环境，可以根据需要定制。

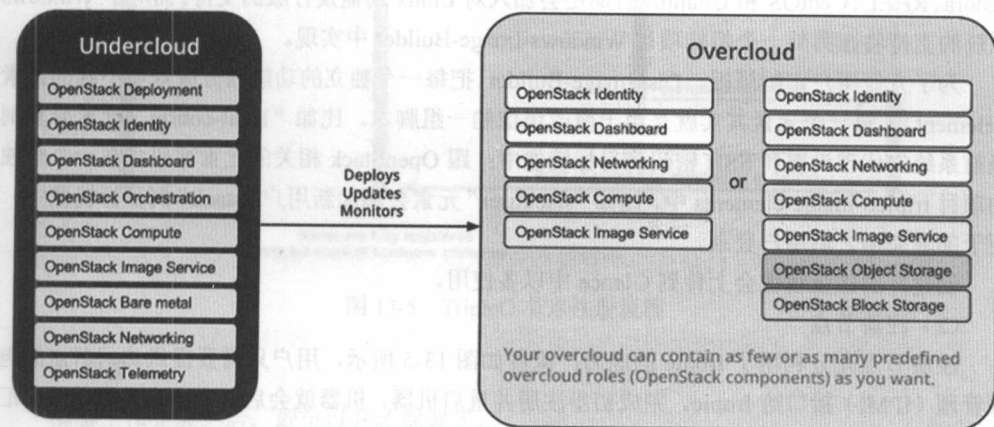


图 13-4 TripleO 功能示意图

底层云利用 Nova 和 Ironic 来管理上层云的物理机节点，用 Neutron 来提供网络环境，用 Glance 来存储部署的磁盘镜像，用 Ceilometer 来收集上层云的统计数据。

1. 安装底层云

用户可以使用 TripleO 的子项目 instack-undercloud 安装底层云。

2. 部署前准备

部署前的准备工作，包括创建磁盘镜像、注册物理机节点、分配角色等。

（1）创建磁盘镜像

TripleO 用 diskimage-builder 来创建磁盘的镜像。

从 OpenStack 部署的角度上看，一个完整的节点（如 Nova 计算节点）包含 3 个部分的内

容：镜像文件、内核和一些常用配置等通用的内容；元数据，如网络等相关信息和服务器地址等，每次部署都会做相应的改变；持久性数据，如数据库或 Swift 这样的存储内容，都会固定存储在另外一处，不会因为重新部署而丢失内容。

DiskImage-Builder 的工作目的就是根据节点的用途（比如用做网络控制节点）制作特定的镜像文件。这样制作镜像的目的就是为了使其既具有共性又有特性，以及可移植性。共性在于通过提前定制一些通用的内容，预安装配置所需服务，从而减少重复的工作。特性在于管理员可以在同样镜像文件的基础上通过更改元数据实现灵活的部署。可移植性在于镜像可以先运行在测试环境中进行检验测试，然后就可以直接部署到生产环节中直接面向客户。

DiskImage-Builder 的工作原理就是利用 chroot 来制作镜像。镜像可以是一个文件系统或者是一个包含文件系统的磁盘镜像文件，通过把镜像格式改成 raw 格式，从而挂载到系统上再开始定制化修改，最后再把镜像转换为 qcow2 等格式。目前已经支持的镜像系统模板有 Fedora、RHEL、CentOS 和 Ubuntu，后期还会加入对 Linux 其他发行版的支持。而对于 Windows 系统的支持将在另外一个姐妹项目 Windows-Image-Builder 中实现。

为了方便用户定制模板，DiskImage-Builder 把每一个独立的功能划分成众多不同的元素（Element），每一个元素其实就是用于修改镜像的一组脚本。比如“local-config”元素会复制当前系统的代理设置和 SSH 密码到目标镜像中；跟 OpenStack 相关的元素都包含在一个单独的项目 tripleo-image-elements 中，比如“stackuser”元素会添加新用户“stack”到目标镜像中，便于部署完毕之后用户登录。

创建好的磁盘镜像会上传到 Glance 中以备使用。

（2）注册节点

注册节点充分利用了 Ironic 的能力，流程如图 13-5 所示，用户只需要提供访问节点的电源管理（IPMI）接口给 Ironic，完成初步注册并重启机器，机器就会启动进入自发现模式，汇报机器的硬件属性细节，最终完成注册。

（3）分配角色

一个云环境会包含多台物理机节点，分别安装不同的服务，比如控制节点、计算节点等，我们称之为不同的角色。与一个角色对应的，包含如下资源。

- 磁盘镜像：镜像中需要包含基础操作系统及对应的软件，比如对计算节点来说，需要有 hypervisor 及 nova-compute 服务等。
- Flavor：对应的硬件需求。计算节点需要有比较多的 CPU 和 RAM 资源。
- 角色节点数量：这个角色包含几个节点。对于一个非 HA 的部署，控制节点数量就是 1，计算节点则需要多个。
- Heat 模板：负责部署完成后的配置工作，保证各个 OpenStack 服务能够正常运行。

所有的这些资源通过 heat 模板联系起来，子项目 tripleo-heat-templates 定义了各个角色的模板。

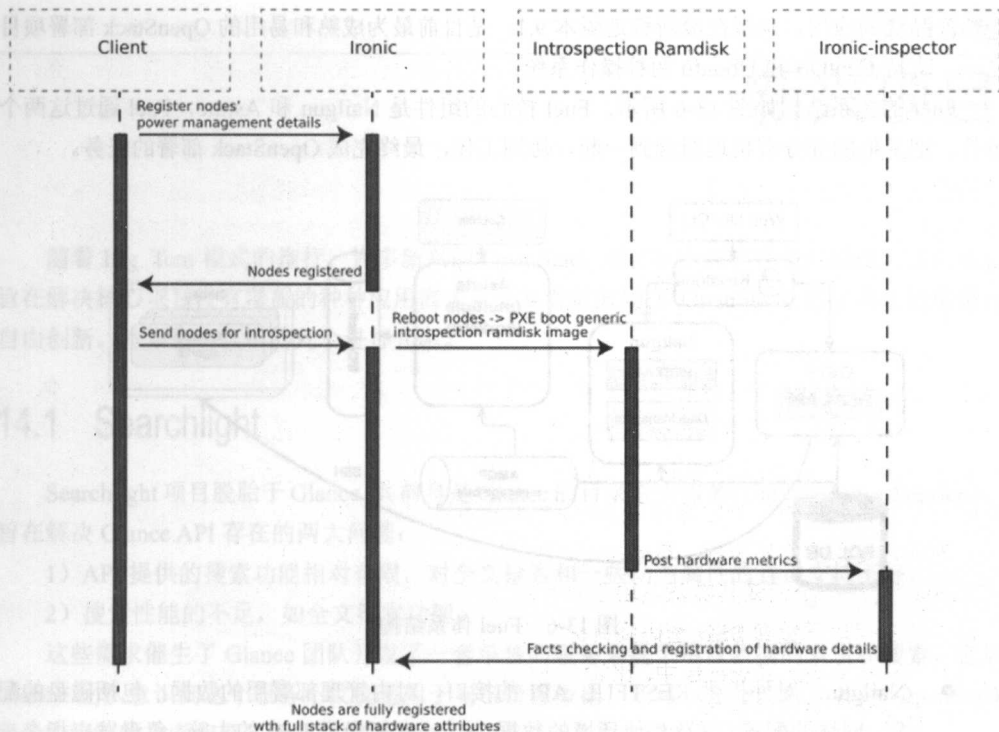


图 13-5 TripleO 节点注册流程

3. 部署

准备工作完成之后，就可以开始部署了。

对于每个角色，Heat 首先需要向 Nova 申请资源，Nova 依赖 IroniC 的 driver 提供一个符合要求的物理节点。

物理节点选定以后，IroniC 设置 PXE 启动节点到一个 RAM DISK，再把从 Glance 中取下来的磁盘镜像写入节点的磁盘里。

节点启动之后，由 OS-Configuration 完成最后的配置工作。OS-Configuration 包括 3 个部分：os-collect-config 负责从 Heat 收集配置信息，写到本地的元数据缓存中，然后调用 os-refresh-config，os-refresh-config 根据元数据来进行系统配置，并且调用 os-apply-config 来生成各个服务需要的配置文件。

13.2.4 Fuel

Fuel 是一个由 Mirantis 主导开发的，图形化的 OpenStack 部署和管理工具，目标是方便的部署、测试以及维护大规模的 OpenStack 云，是 Mirantis OpenStack 的基础，从 2013 年就

开始产品级的应用，到现在最新稳定版本 9.1，是目前最为成熟和易用的 OpenStack 部署项目之一，支持 CentOS 和 Ubuntu 两种操作系统。

Fuel 的体系结构如图 13-6 所示。Fuel 核心的组件是 Nailgun 和 Astute，Fuel 通过这两个组件，把其他的部分有机地组合到一起，协同工作，最终完成 OpenStack 部署的任务。

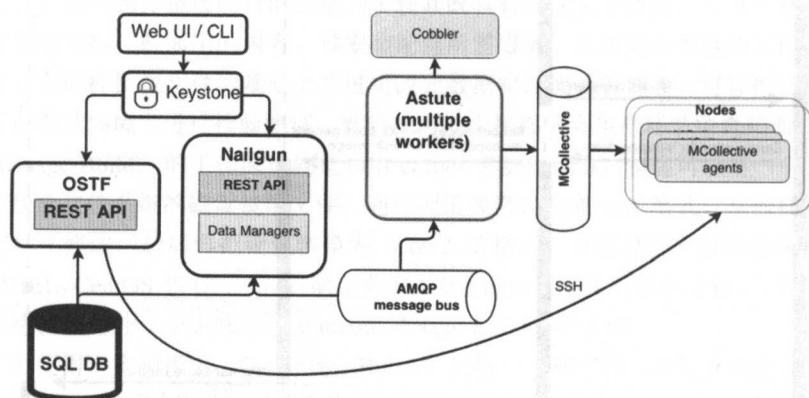


图 13-6 Fuel 体系结构

- Nailgun: 对外提供 RESTFUL API 的接口，对内管理部署用的数据，包括磁盘的配置、网络的配置，以及其他环境的数据。Nailgun 相当于 Fuel 的大脑，负责发出指令，然后由 workers 来完成具体的任务。
- Astute: 是 Nailgun 的 worker，负责与其他服务比如 Cobbler、Puppet 进行交互。接收到 Nailgun 分配到的任务之后，按照私有接口通知其他服务来完成工作。

Fuel UI 是一个功能强大的网页用户界面，用 JavaScript 开发，负责与 Nailgun 的交互，让用户可以以图形界面的方式实现机器的管理、角色分配配置等。用户通过 Fuel UI 的输入会被 Nailgun 保存在后端数据库里，用做后面部署的元数据。

Astute 在接到 Provision（操作系统安装）的命令之后，会根据 Nailgun 提供的每个节点数据来创建对应的 Cobbler system，之后根据 IPMI 的参数来 reboot 节点，Cobbler 根据 Cobbler system 的设置完成操作系统的安装。

Astute 在接到 Deploy（部署 OpenStack）命令之后，根据数据生成配置文件/etc/astute.yml 并通过 Mcollective 上传到被部署节点上。Puppet 解析这个文件，并最终完成部署。具体的 Puppet 模块是由 puppet-ansible 这个项目提供的。

OSTF（OpenStack Testing Framework）用来在完成 OpenStack 部署后做功能验证，检查该部署是否有潜在问题。

Fuel 还集成了 zabbix agent 来监控各个 service 运行时的状态，对于出错的 service 发出警报。Octane 项目提供了 Fuel 主节点及 OpenStack 的升级功能。

详细的信息可以阅读 Fuel 的主页 <http://wiki.openstack.org/wiki/Fuel>。

新兴项目

随着 Big Tent 模式的推行，许多新兴的 OpenStack 项目如雨后春笋般的涌现，这些项目旨在解决核心项目没有覆盖的种种应用需求，开发者被鼓励在 OpenStack 的生态环境中进行自由创新。本章会对其中的几个进行介绍。

14.1 Searchlight

Searchlight 项目脱胎于 Glance，其前身是 Glance 的目录索引服务(Catalog Index Service)，旨在解决 Glance API 存在的两大问题：

- 1) API 提供的搜索功能相对有限，对全文检索和一些动态属性的查询支持不好。
- 2) 搜索性能的不足，如全文检索功能。

这些需求催生了 Glance 团队开发了一套单独的服务来提供镜像数据的索引和搜索。之后 Glance 团队意识到这个服务不仅仅适用于索引镜像数据，还可以通用地解决各种 OpenStack 服务中资源的搜索问题，于是决定把它分离出作为一个独立的项目 Searchlight 来运作。

14.1.1 Searchlight 体系结构

OpenStack 服务以 plugin 的形式添加到 Searchlight 项目中，Searchlight 会根据资源的变化实时更新索引中的数据，对外提供搜索和分析功能的 API，用户访问 API 需要 Keystone 的授权，同时搜索内容也根据租户和 admin 角色做出了严格的区分，确保了数据的安全性。

目前 Searchlight 已经添加了大部分 OpenStack 核心项目的 plugin，比如 Nova、Neutron、Glance、Cinder、Swift 等，并且在不断增加新的服务，用户可以在 Searchlight 中搜索到虚拟机、网络、存储等等资源信息。如图 14-1 所示，一旦用户操作改变了 cloud services 中的资源，searchlight 会通过 notification 感知到这些变化并索引新的资源数据，horizon 和其他客户端可以通过 searchlight API 来查询最新的数据。

同时 Publisher 的功能也在讨论和开发中，希望可以做到一旦资源发生了变化，Searchlight 能够实时地将资源的详细信息推送给感兴趣的服务或用户。

Searchlight 的体系结构如图 14-2 所示。

Searchlight Concept Flow

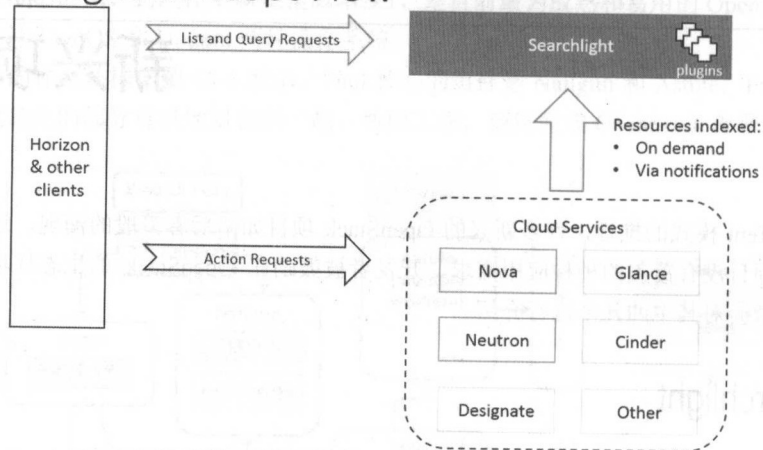


图 14-1 Searchlight 工作流程

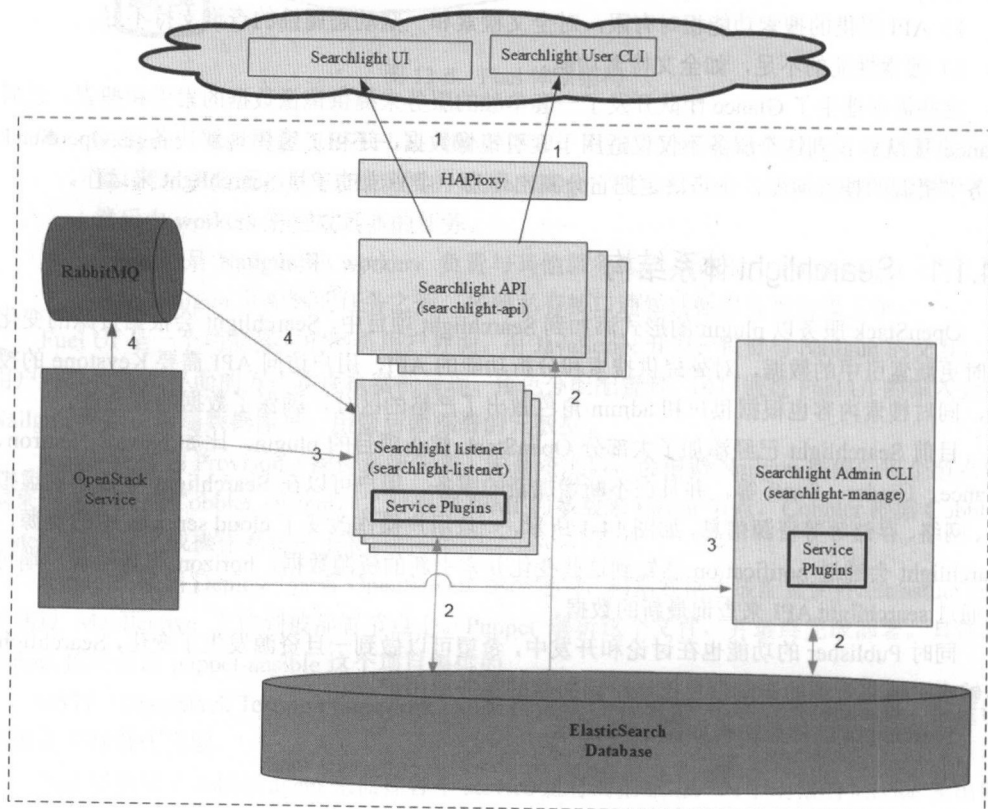


图 14-2 Searchlight 架构图

- UI 和 Client 供外部用户使用，统一通过 Searchlight API 获取搜索结果。
- API 负责读取 Elasticsearch 提供搜索服务，Searchlight 采用了 Elasticsearch query DSL 编写查询语句，用户可以对资源信息进行各种搜索和分析，感兴趣的读者可以查阅 Elasticsearch 文档。
- Listener 监听 RabbitMQ 上的消息通知，如果 OpenStack Service 内发生了资源的变化，相应的通知会发送到 RabbitMQ 上，Listener 会根据通知来更新 Elasticsearch。
- Admin CLI 提供了一系列供 Admin 使用的命令，其中最重要的是对数据库进行初始化，通过调用 OpenStack Service API 重建数据索引。这个命令既可以在部署完 Searchlight 后使用，也可以在数据发生异常时重建数据库。
- HAProxy 配合多个 Searchlight API 进程可以提供高可用性的服务，并且能够灵活地根据负载高低进行水平扩展。

14.1.2 plugin 的开发

在 Searchlight 中索引 Openstack 服务是通过添加 plugin 来实现的，这里以 Neutron Network 为例介绍一下如何开发一个 Searchlight plugin。

(1) 准备 API 和 notification 测试数据

如前所述，Searchlight 获取的 OpenStack 服务数据来自于相关服务 API 和 notification，所以我们可以从收集 API 和 notification 数据开始，这些数据有助于更好地确定 plugin 如何定义 Elasticsearch 数据结构和如何处理 notification，也可以为之后的功能测试提供测试数据。这里展示了一个利用 Neutron API 来获取网络信息的实例程序，根据输出的内容可以来设计 Elasticsearch 索引字段。

```
# list_network.py

import json
import os

from keystoneclient.auth.identity import v2
from keystoneclient import session
from neutronclient.v2_0 import client as nc_20

def get_session():
    username = os.environ['OS_USERNAME']
    password = os.environ['OS_PASSWORD']
    auth_url = os.environ['OS_AUTH_URL']
    tenant_name = os.environ['OS_TENANT_NAME']
    auth = v2.Password(**locals())
    return session.Session(auth=auth)
```

```
nc = nc_20.Client(session=get_session())
networks = nc.list_networks()

print(json.dumps(networks, indent=4, sort_keys=True))
```

运行结果如下：

```
{
  "networks": [
    {
      "admin_state_up": true,
      "availability_zone_hints": [],
      "availability_zones": [
        "nova"
      ],
      "created_at": "2016-04-08T16:44:17",
      "description": "",
      "id": "4d73d257-35d5-4f4e-bc71-f7f629f21904",
      "ipv4_address_scope": null,
      "ipv6_address_scope": null,
      "is_default": true,
      "mtu": 1450,
      "name": "public",
      "port_security_enabled": true,
      "provider:network_type": "vxlan",
      "provider:physical_network": null,
      "provider:segmentation_id": 1053,
      "router:external": true,
      "shared": false,
      "status": "ACTIVE",
      "subnets": [
        "abcc5896-4844-4870-a5d8-6ae4b8edd42e",
        "ea47304e-bd54-4337-901a-1eb5196ea18e"
      ],
      "tags": [],
      "tenant_id": "fa1537e9bda9405891d004ef9c08d0d1",
      "updated_at": "2016-04-08T16:44:17"
    }
  ]
}
```

Searchlight 提供了一个简单的脚本 `searchlight/test-scripts/listener.py` 用于监听消息总线上的通知，用户可以用它来获取资源变化的通知内容作为测试数据。

(2) 定义 Index 类

plugin 代码主要包括两部分，即 Index 类和 Notification Handler。Index 类主要用来定义和

ElasticSearch index 有关的各种配置和行为, 主要包括 mapping、权限控制等。

```
# searchlight/elasticsearch/plugins/neutron/networks.py

from searchlight.common import resource_types
from searchlight.elasticsearch.plugins import base

# index 类需要继承 IndexBase 基类
class NetworkIndex(base.IndexBase):
    NotificationHandlerCls = notification_handlers.NetworkHandler

    def get_mapping(self):
        return {
            # dynamic 用来控制 Elasticsearch 如何处理没有定义的字段信息
            # False 代表不索引 mapping 中没有定义的字段
            'dynamic': False,
            # properties 定义了需要索引的字段和类型
            'properties': {
                'admin_state_up': {'type': 'boolean'},
                'availability_zone_hints': {'type': 'string',
                                           'index': 'not_analyzed'},
                'availability_zones': {'type': 'string',
                                       'index': 'not_analyzed'},
                'created_at': {'type': 'date'},
                'description': {'type': 'string'},
                'id': {'type': 'string', 'index': 'not_analyzed'},
                'ipv4_address_scope': {'type': 'string',
                                       'index': 'not_analyzed'},
                'ipv6_address_scope': {'type': 'string',
                                       'index': 'not_analyzed'},
                'mtu': {'type': 'integer'},
                'name': {
                    'type': 'string',
                    'fields': {
                        'raw': {'type': 'string', 'index': 'not_analyzed'}
                    }
                },
                'port_security_enabled': {'type': 'boolean'},
                .....
            },
            # _meta 记录 properties 中和其他资源关联的字段
            "_meta": {
                "project_id": {
                    "resource_type": resource_types.KEYSTONE_PROJECT
                },
            },
        }
```



```

        "tenant_id": {
            "resource_type": resource_types.KEYSTONE_PROJECT
        }
    }
}

```

mapping 里包括和 Elasticsearch index 有关的各种配置，其中最重要的是 properties 字段，这个字典定义了 Elasticsearch 索引的结构，大多数字段会和 Neutron API 返回的字段相同，这样用户的搜索结果就能够和 API 返回结果保持一致。

对于字符串字段，index 属性会决定 Elasticsearch 是否对字段内容做分词并建立倒排索引，这是搜索引擎针对全文检索需求常用的处理方法。对于某些只希望匹配完整值的字段，我们可以通过设置 index 属性为 not_analyzed 来避免分词以节省不必要的开销。

```

# searchlight/elasticsearch/plugins/neutron/networks.py

from searchlight.common import resource_types
from searchlight.elasticsearch.plugins import base

class NetworkIndex(base.IndexBase):
    NotificationHandlerCls = notification_handlers.NetworkHandler

    def _get_rbac_field_filters(self, request_context):
        """返回的 filter 会被注入到搜索语句中，document type 过滤默认会
        添加到 filter 列表中
        """
        return [{
            'bool': {
                'should': [
                    {'term': {'tenant_id': request_context.owner}},
                    {'terms': {'members': [request_context.owner, '*']}},
                    {'term': {'router:external': True}},
                    {'term': {'shared': True}}
                ]
            }
        ]
}

```

在 plugin 中可以设置 filter 来对搜索结果做权限控制（Role Based Access Control），这里网络 plugin 设置了一个 bool 型的组合 filter，should 表示满足列表中任意一个 filter 的记录都会被添加到返回的搜索结果之中，属于搜索请求租户的网络，通过 Neutron 网络 RBAC policy 分享的网络以及分享给所有租户的网络，这里的 filter 语法来自于 Elasticsearch query DSL。通过 plugin 的 RBAC filter 和 oslo policy 相结合，Searchlight 提供了完备的搜索权限控制以保证数据安全。

（3）定义 notification handler 类

除了用定义 Index 类来设置 Elasticsearch 交互的细节,我们还需要一个 Notification Handler 来处理监听通知。

```
# searchlight/elasticsearch/plugins/neutron/notification_handler.py

class NetworkHandler(base.NotificationBase):
    @classmethod
    def _get_notification_exchanges(cls):
        return ['neutron']

    def get_event_handlers(self):
        # 定义需要监听的通知事件和处理事件的回调函数
        return {
            'network.create.end': self.create_or_update,
            'network.update.end': self.create_or_update,
            'network.delete.end': self.delete,
            'rbac_policy.create.end': self.rbac_create,
            'rbac_policy.delete.end': self.rbac_delete
        }

    def create_or_update(self, payload, timestamp):
        network_id = payload['network']['id']
        LOG.debug("Updating network information for %s", network_id)
        # 对通知进行加工, 补全网络信息, 计算版本号
        network = serialize_network(payload['network'])
        version = self.get_version(network, timestamp)
        # 将转化后的文档存入 Elasticsearch
        self.index_helper.save_document(network, version=version)
```

当用户在 Neutron 里建立或更新了一个网络时,对应的回调函数会被执行,对通知的内容(payload)做一些数据加工,将其转化为一个与 plugin mapping 相对应的字典,然后调用 index_helper 类的方法将这个文档存入 Elasticsearch。为了确保数据插入的准确性,plugin 通常会根据通知的内容计算出一个版本号,如果更新的资源数据并不比库中存储的新,这次更新就会以失败而告终。index_helper 是一个与 Elasticsearch 做交互的工具类,在内部使用 Elasticsearch Python Client 来进行增删改查的工作。

(4) 注册 plugin

Searchlight 使用 stevedore 来读取所有可用的 plugin,需要将 plugin 加入到 setup.cfg 文件 entry_points 里的 index_backend driver 中,如下所示:

```
searchlight.index_backend =
os_neutron_network=searchlight.elasticsearch.plugins.neutron.networks:NetworkIndex
```

14.2 Watcher

Watcher 项目主要关注 Cloud 的资源优化。Cloud 本质就是分配一定资源来完成特定的工作负载，但工作负载是动态变化的，也就要求对原先分配的资源进行调整与之适应，这个过程就是资源优化。目前 OpenStack 的资源优化主要依靠人工监控并调整，运营成本太高，Watcher 的目标，就是使资源优化可以自动进行，降低运营的成本。

Watcher 由 B-com、IBM 和 Intel 联合创建，并于 2016 年 5 月成为 OpenStack 的 Big Tent 项目。目前，Watcher 历经了小规模部署、可扩展性测试，已经开始在真正的生产环境中使用。很多使用 OpenStack 的公司，都对 Watcher 表现出了极大的兴趣。

Watcher 主要完成了以下 3 个任务：

- 1) 为基于 OpenStack 的多租户 Cloud，提供了灵活的、可扩展的资源优化服务。
- 2) 提供了完整的框架，可以用来实现各种各样的优化目标，比如能耗优化、负载优化、性能优化等。
- 3) 系统架构级别支持插件，支持第三方基于新的性能指标 (Metrics) 开发新的优化算法 (Strategy)。

这里是一些具体的资源优化的例子：

- 1) 基于 CPU 利用率或者 IO 吞吐，进行虚拟机的动态迁移。
- 2) 调整虚拟机的大小。
- 3) 基于负载，动态调整计算节点的能耗。

通过这些优化，可以高效地均衡负载，降低数据中心的运营成本，并减少人工干预。

Watcher 作为 Big Tent 的子项目，常用功能依赖于其他的项目。图 14-3 所示为 Watcher 在 OpenStack 生态系统和其他项目的关系。

首先，Watcher 本身用到了 Oslo 的很多库，比如 log、config 等，并且依赖 Keystone 提供鉴别机制。其次，Watcher 的优化算法需要很多性能数据，比如 CPU 使用率，作为输入。而这些性能数据，是由 Ceilometer 或者 Monasca 来提供的。最后，Watcher 的优化算法最终会生成执行计划 (Action Plan)，它们又通过调用各个项目的 API 来实现，比如 Nova 的动态迁移、Ironic 的电源管理。

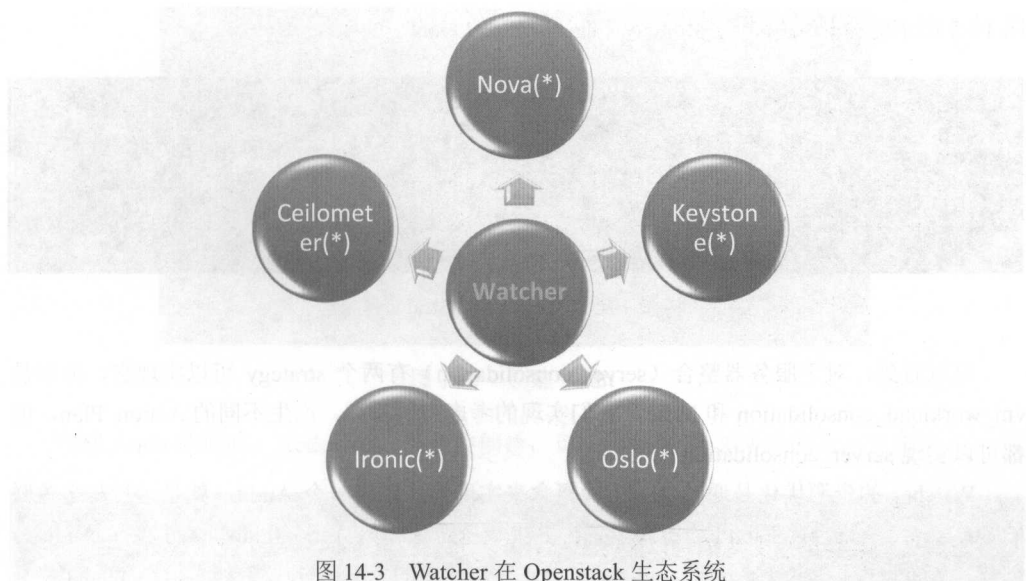


图 14-3 Watcher 在 Openstack 生态系统

14.2.1 Watcher 使用

Watcher 的安装部署和其他项目一样，没有太大的区别，目前已经集成到 Devstack 和 Ansible 中。这里主要介绍 Watcher 的使用。

Watcher 的插件已被集成到 Horizon 中，可以通过 Horizon 的 GUI 来使用 Watcher。下面介绍一下如何通过命令行来使用 Watcher。

使用 Watcher 前，先了解一下 Watcher 的两个重要概念：Strategy 和 Goal。每一个 Strategy 本质上就是一个优化算法，来实现一个特定的目标（Goal）。多个 Strategy 可以针对不同的考虑来实现同一个 Goal。图 14-4 所示的命令可以查看所有的 Goal。

```
root@jzfzlr04h07:~/edwin/watcher-pkb# watcher goal list
```

UUID	Name	Display name
f5cfa5ae-4b1d-40d7-a651-dac3c585fa47	dummy	Dummy goal
03e66f5b-f43a-4d9e-92f2-4adb07c5dcd5	workload_balancing	Workload Balancing
0b1bd544-a3fe-44ba-af9d-bb7ed687d563	server_consolidation	Server Consolidation
2da53310-0b10-44b3-a4ec-9b078893141f	thermal_optimization	Thermal Optimization
75dd46fe-4f3f-4db7-a4ba-16b4d4643e89	airflow_optimization	Airflow Optimization
8342273e-086d-4f90-987f-c51375dd4f84	unclassified	Unclassified

图 14-4 Watcher 的 Goal

可以看到这里面有各种各样的 Goal：负载均衡、服务器整合、温度优化和进风量优化。Goal 本身只是一个目标的描述，只有当存在某个 Strategy 来实现它时，这个 Goal 才有意义。

图 14-5 所示的命令可以查看 Strategy 和它们对应的 Goal。

```
root@jflr04h07:~/edwin/watcher-pkb# watcher strategy list
```

UUID	Name	Display name	Goal
2a7ad582-63df-4d78-adb4-4e3281a5d23b	dummy	Dummy strategy	dummy
7a97ab97-3b0f-4b81-ace3-4c666bc29404	dummy with scorer	Dummy Strategy using sample Scoring Engines	dummy
beb941a9-c258-43e6-9b8d-b5e75044c81b	outlet temperature	Outlet temperature based strategy	thermal_optimization
f44e80e6-30ba-4005-94da-d15e13088e65	vm workload consolidation	VM Workload Consolidation Strategy	server_consolidation
3ecc0e79-e5f8-4bbe-9770-3dd5b4b38d4a	basic	Basic offline consolidation	server_consolidation
0c681a3e-f381-4d53-8eb4-1ad41e04ba3a	workload stabilization	Workload stabilization	workload_balancing
59e370f1-fcf7-4b47-acc4-3b0296c394ea	workload balance	Workload Balance Migration Strategy	workload_balancing
0cb96ef2-824e-4148-870a-9cb8df252abb	uniform airflow	Uniform airflow migration strategy	airflow_optimization

图 14-5 Watcher 的 Strategy

可以看到，对于服务器整合（server_consolidation）有两个 strategy 可以实现它，分别是 vm_workload_consolidation 和 basic。它们实现的考虑有所不同，产生不同的 Action Plan，但都可以实现 server_consolidation 的 Goal。

Watcher 的资源优化是通过 Audit 的概念来实现的。创建一个 Audit，就是运行与之关联的 Strategy，考察当前 Cloud 的资源配置，并生成资源优化的 Action Plan。为了便于 Audit 的创建，Watcher 还引入了 Audit template 来定义经常使用的 Audit 的共同参数，比如 Goal 等。这样，管理员就可以先创建一个 Audit template，然后随时创建出一个相关的 Audit 进行资源优化。

Audit template 对应的 Goal 必须在创建时指定，而对应的 Strategy 可以由静态或动态的方法指定。静态方法是在 Audit template 创建时指定需要的 Strategy，这样的好处是可以为 Strategy 输入参数。动态方法是在 Audit 创建时，由系统选择需要的 Strategy。图 14-6 所示的命令就创建了一个实现温度优化的 Audit template，并且静态选择 Strategy。

```
root@jflr04h07:~/edwin/watcher-pkb# watcher audittemplate create -s outlet_temperature -s threshold=36.0 outlet-temp thermal_optimization
```

Field	Value
UUID	390c4878-73de-42e2-bdce-894ae4a1438d
Created At	2016-11-06T01:20:29.772139+00:00
Updated At	None
Deleted At	None
Description	None
Host Aggregate ID or Name	None
Name	outlet-temp
Extra	{u'threshold': 36.0}
Goal	thermal_optimization
Strategy	outlet_temperature

图 14-6 创建 Audit template

这个 Audit template 选定了 outlet_temperature 作为它的 Strategy，并且给 Strategy 指定了一个参数 threshold=36.0。

有了 Audit template，就可以由它来创建一个 Audit。每一次 Audit 的创建，都会执行对应的 Strategy，对现有的 Cloud 进行一次资源优化考察，并生产对应的 Action Plan。其命令如图 14-7 所示。


```

root@jzfzlr04h07:~/edwin/watcher-pkb# watcher audit create -a outlet-temp
+-----+-----+
| Field | Value |
+-----+-----+
| UUID | 8b7e714e-ac89-490b-92d5-db898a037d7e |
| Created At | 2016-11-07T01:39:16.082846+00:00 |
| Updated At | None |
| Deleted At | None |
| Deadline | None |
| State | PENDING |
| Audit Type | ONESHOT |
| Parameters | {u'threshold': 35.0} |
| Interval | None |
| Host Aggregate ID or Name | None |
| Goal | thermal_optimization |
| Strategy | outlet_temperature |
+-----+-----+

```

图 14-7 创建 Audit

创建 Audit 的同时，Action Plan 也会被创建，可以用如图 14-8 所示的命令来查看。

```

root@jzfzlr04h07:~/edwin/watcher-pkb# watcher actionplan list
+-----+-----+-----+-----+-----+-----+
| UUID | Audit | State | Updated At | Global efficacy |
+-----+-----+-----+-----+-----+-----+
| 8fa5ff01-d038-4cb0-a283-37995f974ad1 | 8b7e714e-ac89-490b-92d5-db898a037d7e | SUCCEEDED | 2016-11-07T01:39:17+00:00 | None |
+-----+-----+-----+-----+-----+-----+

```

图 14-8 自动创建的 actionplan

然后可以用如图 14-9 所示的命令执行这个 Action Plan，实质上就是执行几个动态迁移的 Action，降低特定计算节点的温度，达到资源优化的目的。

```

root@jzfzlr04h07:~/edwin/watcher-pkb# watcher actionplan start 8fa5ff01-d038-4cb0-a283-37995f974ad1

```

图 14-9 执行生成的 Action plan

如上所述即为 Watcher 的基本使用过程：找到满足自己要求的 Goal，然后创建 Audit template；然后由 template 创建 Audit，这个过程自动执行对应的 Strategy，并生成资源优化的 Action Plan；最后执行 Action Plan 完成优化。可以看出，Watcher 和 Cloud 里面的 VM scheduler 有很大不同。Watcher 是在 Scheduler 完成 VM 的部署后，进行再平衡。它们是高度互补的技术，不会相互替代。

14.2.2 Watcher 体系结构

为了灵活性以及可扩展，Watcher 采用了如图 14-10 的架构。

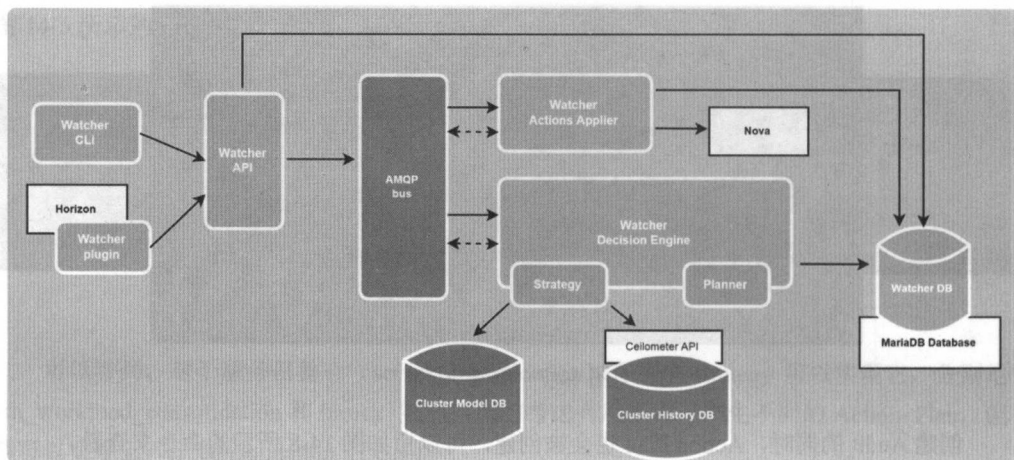


图 14-10 Watcher 的架构图

位于 Watcher 系统外部的是 CLI（命令行），和 Horizon 的插件，用户主要通过这两个模块调用 API 来使用 Watcher。

Watcher 系统本身，逻辑上分为 3 部分，即 Watcher API、Applier 和 Decision Engine。

1) API: 与其他 OpenStack 项目一样，Watcher 也实现了一套 RESTful API 将自己的功能提供给外部使用。

2) Decision Engine: Watcher 的核心，以实现 Audit 对应的 Goal 为目的，计算出资源优化所需的 Action Plan。首先，读取 Audit 对应的 Goal，找出适合的所有 Strategy 列表。如果没有指定 Strategy，Decision Engine 就选取一个最合适的，并通过 Stevedore 动态加载。最后，Strategy 被运行，根据当前集群的状态和要实现的 Goal，生成最终的 Action Plan。

3) Applier: 负责执行 Decision Engine 生成的 Action Plan。从 Decision Engine 获取要执行的 Action Plan 的 UUID，并到数据库中查询 Action Plan 的详细信息，然后遍历执行其中的每个 Action。一般来说，每个 Action 都会通过 Keystone API 得到一个令牌（token），然后向某个 OpenStack 项目的 RESTful API 发出请求，执行特定的操作，比如 Nova 的动态迁移。

除了这些逻辑模块，Watcher 还有一些内部支持模块，比如 AMQP bus、Cluster History DB 和 Watcher DB。

1) AMQP bus: 处理 Watcher 内部模块的异步通信。比如 Decision Engine 需要 Applier 来执行一个 Action Plan 时，就是通过 AMQP bus 将 UUID 传递给 Applier。Watcher API 需要处理一个创建 Audit 的请求时，也是通过 AQMP bus 将参数传递给 Decision Engine 的。

2) Cluster History DB: Cluster History 包含了所有收集的指标（metrics）和事件，代表了当前集群的资源配置情况。这些数据存储在数据库中，能够被 Strategy 使用来找到最优的资源优化方案。

3) Watcher DB: 这个数据库存储 Watcher 的所有对象，包括 Goal、Strategy、Audittemplate、

Audit、Actionplan 和 Action。

14.2.3 strategy 的开发

如前所述，Watcher 的架构支持插件，方便用户自行开发代码，满足特定的需求。目前，可以通过开发插件，来实现新的 Goal、Strategy、Action 等。这里介绍如何实现新的 Strategy，Goal 和 Action 的实现也是大同小异。

1) 选择正确的 Strategy 基类，派生出新的 Strategy 子类，可以利用的基类在 watcher/decision_engine/strategy/strategies/base.py 中。

```
@six.add_metaclass(abc.ABCMeta)
class UnclassifiedStrategy(BaseStrategy):
    @classmethod
    def get_goal_name(cls):
        return "unclassified"

@six.add_metaclass(abc.ABCMeta)
class ServerConsolidationBaseStrategy(BaseStrategy):
    @classmethod
    def get_goal_name(cls):
        return "server_consolidation"

@six.add_metaclass(abc.ABCMeta)
class ThermalOptimizationBaseStrategy(BaseStrategy):
    @classmethod
    def get_goal_name(cls):
        return "thermal_optimization"

@six.add_metaclass(abc.ABCMeta)
class WorkloadStabilizationBaseStrategy(BaseStrategy):
    @classmethod
    def get_goal_name(cls):
        return "workload_balancing"
```

这些 Strategy 的基类只定义了一个类函数，返回对应的 Goal 名称。虽然简单，但是建立了从 Strategy 到 Goal 的映射关系。我们可以根据 Goal 选择需要的基类，比如 Consolidation、Thermal 或者 Workload。如果都不满足，可以选择 UnclassifiedStrategy。

2) 从 Strategy 基类派生出新的子类之后，需要实现下列函数。

- get_name 类函数：返回新 Strategy 独有的名称。
- get_diaplay_name 类函数：返回翻译过的 strategy 的显示名。
- get_translatable_display_name 类函数：返回 strategy 的英文显示名。
- execute 抽象函数：返回资源优化的最终方案，本质上是 BaseSolution 的一个实例。

它会产生最终的 Action Plan。

这里是一个实现新的 Strategy 的简单示例，具体代码如下：

```
class NewStrategy(base.UnclassifiedStrategy):

    def __init__(self, osc=None):
        super(NewStrategy, self).__init__(osc)

    def execute(self, original_model):
        self.solution.add_action(action_type="nop",
                                input_parameters=parameters)
        # Do some more stuff here ...
        return self.solution

    @classmethod
    def get_name(cls):
        return "new_strategy"

    @classmethod
    def get_display_name(cls):
        return _("New strategy")

    @classmethod
    def get_translatable_display_name(cls):
        return "New strategy"
```

3) 定义 Strategy 的参数。

对于新的 Strategy，可以定义一些参数规范。这样管理员在创建 Audit 时，可以输入 Strategy 参数，而 Strategy 的 execute 函数则可以读取这些参数，用于计算资源优化方案。比如，Strategy 可以定义一些阈值（threshold），方便管理员对资源优化进行动态调优。

定义参数，只要实现 Strategy 的 get_schema 函数，返回 jsonschema 格式的参数规范。规范中，一般需要提供参数名称、描述、类型、最大值、最小值和默认值。后面 3 个是可选的，但建议提供默认值，否则当管理员创建 Audit 时没有输入参数时，会产生异常错误。

这里是一个定义了两个参数的 Strategy 示例，具体代码如下：

```
class DummyStrategy(base.DummyBaseStrategy):
    @classmethod
    def get_schema(cls):
        return {
            "properties": {
                "para1": {
                    "description": "number parameter example",
                    "type": "number",
```

```

        "default": 3.2,
        "minimum": 1.0,
        "maximum": 10.2,
    },
    "para2": {
        "description": "string parameter example",
        "type": "string",
        "default": "hello",
    },
},
}

```

在新 Strategy 的 execute 函数中，可以这样访问这两个参数：

```

class DummyStrategy(base.DummyBaseStrategy):
    def execute(self):
        para1 = self.input_parameters.para1
        para2 = self.input_parameters.para2

        if para1 > 5:
            ...

```

4) 添加新的插件入口点 (Entry Point)。

为了使 Watcher 加载新创建的 Strategy，必须用它的名字来注册一个入口点，这样 stevedore 在 Watcher 启动时就可以加载它。下面是 setup.cfg 中的一个示例，具体代码如下：

```

watcher_strategies =
    dummy =
watcher.decision_engine.strategy.strategies.dummy_strategy:DummyStrategy

```


博文视点诚邀精锐作者加盟

《C++Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博古通今、文圆质方、视角独特、点石成金之计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于IT专业出版之巅。

英雄帖

江湖风云起,代有才人出。

IT界群雄并起,逐鹿中原。

博文视点诚邀天下技术英豪加入,

指点江山,激扬文字

传播信息技术,分享IT心得

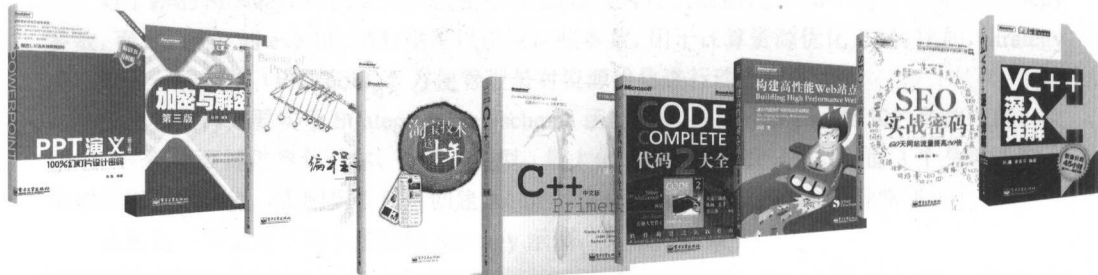
• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照IT技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身定制写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn



@博文视点Broadview



微信公众账号 博文视点Broadview



博文视点精品图书展台

专业典藏



移动开发



大数据 · 云计算 · 物联网



数据库



Web 开发



程序设计



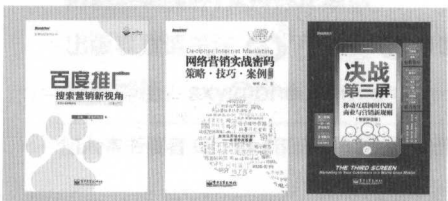
软件工程



办公精品



网络营销



好书力荐



拒绝堆砌臃肿,支持纯正原创

出版事宜请关注  新浪微博 @ 半亩方塘 _
weibo.com

投稿邮箱: sxy@phei.com.cn

加入本书读者 QQ 群 465398813,以书会友,资源共享!

OpenStack设计与实现 (第2版)

Since its introduction in 2010, OpenStack has gained tremendous momentum with thousands of developers and a couple of thousand deployments. Today, OpenStack is considered the de-facto open source cloud Infrastructure-as-a-Service provisioning software for companies including Baidu, the European Organization for Nuclear Research (CERN), HP, Huawei, IBM, Intel, and Walmart, as well as many more Fortune 500 companies.

It is with immense joy that we present this book to the Chinese community. I hope you find it compelling and its content helps increase China's influence in OpenStack development, easing adoption and offering cost savings for new economic endeavors.

Imad Sousou

Intel Vice President and General Manager, Open Source Technology Center
Platinum Board Member, OpenStack Board of Directors



博文视点Broadview



@博文视点Broadview



策划编辑：孙学瑛
责任编辑：徐津平
封面设计：李 玲

上架建议：云计算与大数据

ISBN 978-7-121-31199-4



9 787121 311994 >

定价：99.00元